BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images,
Please do not report the images to the
Image Problem Mailbox.

EP 23632 (5)

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46 — A1

(11) International Publication Number: WO 97/22928

(43) International Publication Date: 26 June 1997 (26.06.97)

(21) International Application Number: PCT/US96/20536

(22) International Filing Date: 17 December 1996 (17.12.96)

(30) Priority Data:
08/579,464 20 December 1995 (20.12.95) US

(71) Applicant: OBJECT TECHNOLOGY LICENSING CORP., doing business as OTLC [US/US]; 10355 N. De Anza Boulevard, Cupertino, CA 95014 (US).

(72) Inventors: ATSATT, Bryan, P.; 3545 Altamont Way, Redwood City, CA 94062 (US). RADOVANCEVICH, Michael, P.; 210 West Bergen Court, Foxpoint, WI 53217 (US). THAKKAR, Hemantkumar, A.; 513 Penitencia Street #2, Milpitas, CA 95035 (US).

(74) Agents: BYORICK, Michael et al.; Object Technology Licensing Corp., 10355 N. De Anza Boulevard, Cupertino, CA 95014 (US).

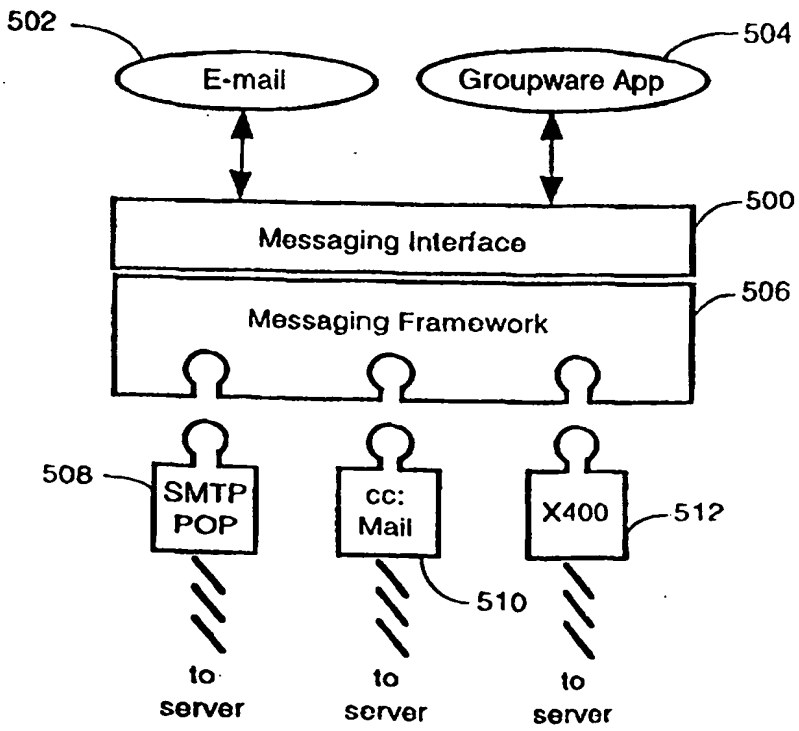
(81) Designated States: CA, JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: AN OBJECT ORIENTED PROGRAMMING BASED MESSAGING INTERFACE SYSTEM AND METHOD

(57) Abstract

An object oriented programming (OOP) based messaging interface method for use between a client application and a messaging service on a computer system is provided. The method includes providing: a message class which has data members and member functions related to a message in the messaging service, a message bin class which has data members and member functions related to holding the message, and a message bin name class which has data members and member functions related to identifying a message bin object instantiated from the message bin class. Subsequently, a messaging interface object for the message is generated as an instance of one or more of the provided classes. This generated messaging interface object furnishes the client application with protocol independent access to the messaging service. In addition, a storage device readable by a computer system for implementing the OOP-based messaging interface method and an OOP-based messaging interface are provided.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	GB	United Kingdom	MW	Malawi
AT	Austria	GE	Georgia	MX	Mexico
AU	Australia	GN	Guinea	NE	Niger
BB	Barbados	GR	Greece	NL	Netherlands
BE	Belgium	HU	Hungary	NO	Norway
BF	Burkina Faso	IE	Ireland	NZ	New Zealand
BG	Bulgaria	IT	Italy	PL	Poland
BJ	Benin	JP	Japan	PT	Portugal
BR	Brazil	KE	Kenya	RO	Romania
BY	Belarus	KG	Kyrgystan	RU	Russian Federation
CA	Canada	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic			SE	Sweden
CG	Congo	KR	Republic of Korea	SG	Singapore
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LR	Liberia	SZ	Swaziland
CS	Czechoslovakia	LT	Lithuania	TD	Chad
CZ	Czech Republic	LU	Luxembourg	TG	Togo
DE	Germany	LV	Latvia	TJ	Tajikistan
DK	Denmark	MC	Monaco	TT	Trinidad and Tobago
EE	Estonia	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	UG	Uganda
FI	Finland	ML	Mali	US	United States of America
FR	France	MN	Mongolia	UZ	Uzbekistan
GA	Gabon	MR	Mauritania	VN	Viet Nam

AN OBJECT ORIENTED PROGRAMMING BASED MESSAGING INTERFACE SYSTEM AND METHOD

Field Of The Invention

The present invention relates generally to a messaging interface to an electronic messaging system for a communication network. More particularly, the present invention relates to such a messaging interface as implemented in an object oriented programming based environment.

Background of the Invention

A variety of different messaging systems are known which are used to transfer data between different users connected to a common network. For example, a number of electronic mail systems are used to exchange mail message between human users typically connected to some form of a communication network via a personal computer. Use of such messaging systems are increasingly used in a number of environments over a wide variety of different type of networks.

The types of users connected to the network are also increasingly adding complexity to the network. For example, one user may be connected to the network via one type of computing platform while another user may use a different type of computing platform. This places an increased burden on the messaging systems to handle messages generated using different software and hardware having different messaging protocols.

Moreover, multiple network systems are being increasingly tied together (e.g., the Internet), effectively forming larger networks connecting systems using even more disparate hardware and software. The messaging systems may be either directly connected to the system using a gateway to allow messages to be routed between the different systems, or indirectly connected (i.e., messaging systems which communicate through a third system and which are not directly connected to the same third system).

Current messaging systems are unable to efficiently handle messaging between the various types of user systems. As a result, the type of data which may be transferred must be limited to common

formats (e.g., ASCII text) to be handled by the different system. The data to be transferred across such heterogeneous environments looses fidelity as a result of translations and transformations to the lowest common denominator capable of being handled by the different systems. Proper messaging over such disparate systems is also hindered by different message protocols. For example, there may be several ways to address a single recipient, adding still further complexity to the system.

In order to address the above problems, attempts have been made to provide application programs capable of operating on different user systems and freely exchanging messages therebetween. Such approaches attempt to specifically address the different types of systems being used to provide messaging capabilities. Another approach is to standardize or limit the types of messaging systems used on a network. This, however, limits the functionality of the network and the messaging systems.

Object oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design and development, electronic messaging systems, like e-mail, will need to be adapted to make use of the benefits of OOP. A need exists for these principles of OOP to be applied to a messaging interface of an electronic messaging system such that a set of OOP classes and objects for messaging interface can be provided.

OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the program. An object is a software package that contains both data and a collection of related structures and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

OOP components, in general, are reusable software modules which present an interface that conforms to an object model and which are accessed at run-time through a component integration architecture.

A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each other's capabilities or functions. This is generally done by assuming a common component object model on which to build the architecture.

It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

OOP allows the programmer to create an object that is a part of another object. For example, the object representing a piston engine is said to have a composition-relationship with the object representing a piston. In reality, a piston engine comprises a piston, valves and many other components; the fact that a piston is an element of a piston engine can be logically and semantically represented in OOP by two objects.

OOP also allows creation of an object that "depends from" another object. If there are two objects, one representing a piston engine and the other representing a piston engine wherein the piston is made of ceramic, then the relationship between the two objects is not that of composition. A ceramic piston engine does not make up a piston engine. Rather it is merely one kind of piston engine that has one more limitation than the piston engine; its piston is made of ceramic. In this case, the object representing the ceramic piston engine is called a derived object and it inherits all of the aspects of the object representing the piston engine and adds further limitation or detail to it. The object representing the ceramic piston engine "depends from" the object representing the piston engine. The relationship between these objects is called inheritance.

When the object or class representing the ceramic piston engine inherits all of the aspects of the objects representing the piston engine, it inherits the thermal characteristics of a standard piston defined in the piston engine class. However, the ceramic piston engine object overrides these thermal characteristics with those specific to a ceramic piston which are typically different from those associated with a metal piston. It skips over the original and uses new functions related to ceramic pistons. Different kinds of piston engines will have different characteristics, but may have the same underlying functions associates with it (e.g., how many pistons in the engine, ignition sequences,

lubrication, etc.). To access each of these functions in any piston engine object, a programmer would call the same functions with the same names, but each type of piston engine may have different/overriding implementations of functions behind the same name. This ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. In fact, our logical perception of the reality is the only limit on determining the kinds of things that can become objects in object-oriented software. Some typical categories are as follows:

- Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control system.
- Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.
- An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.
- An object can represent user-defined data types such as time, angles, and complex numbers, or point on the plane.

With this enormous capability of an object to represent just about any logically separable matters, OOP allows the software developer to design and implement a computer program that is a model of some aspects of reality, whether that reality is a physical entity, a process, a system, or a composition of matter. Since the object can represent anything, the software developer can create an object which can be used as a component in a larger software project in the future.

If 90% of a new OOP software program consists of proven, existing components made from preexisting reusable objects, then only the remaining 10% of the new software project has to be written and tested from scratch. Since 90% already came from an inventory of extensively tested reusable objects, the potential domain from which an error could originate is 10% of the program. As a result, OOP enables

software developers to build objects out of other, previously built, objects.

This process closely resembles complex machinery being built out of assemblies and sub-assemblies. OOP technology, therefore, makes software engineering more like hardware engineering in that software is built from existing components, which are available to the developer as objects. All this adds up to an improved quality of the software as well as an increased speed of its development.

Programming languages are beginning to fully support the OOP principles, such as encapsulation, inheritance, polymorphism, and composition-relationship. With the advent of the C++ language, many commercial software developers have embraced OOP. C++ is an OOP language that offers a fast, machine-executable code. Furthermore, C++ is suitable for both commercial-application and systems-programming projects. For now, C++ appears to be the most popular choice among many OOP programmers, but there is a host of other OOP languages, such as Smalltalk, common lisp object system (CLOS), and Eiffel. Additionally, OOP capabilities are being added to more traditional popular computer programming languages such as Pascal.

The benefits of class libraries can be summarized, as follows:

- Objects and their corresponding classes break down complex programming problems into many smaller, simpler problems.
- Encapsulation enforces data abstraction through the organization of data into small, independent objects that can communicate with each other. Encapsulation protects the data in an object from accidental damage, but allows other objects to interact with that data by calling the object's member functions and structures.
- Subclassing and inheritance make it possible to extend and modify objects through deriving new kinds of objects from the standard classes available in the system. Thus, new capabilities are created without having to start from scratch.
- Polymorphism and multiple inheritance make it possible for different programmers to mix and match characteristics of many different classes and create specialized objects that can still work with related objects in predictable ways.
- Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.

These libraries of reusable classes are useful in many situations, but they also have some limitations. For example:

- Complexity. In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.
- Flow of control. A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The programmer has to decide which functions to call at what times for which kinds of objects.
- Duplication of effort. Although class libraries allow programmers to use and reuse many small pieces of code, each programmer puts those pieces together in a different way. Two different programmers can use the same set of class libraries to write two programs that do exactly the same thing but whose internal structure (i.e., design) may be quite different, depending on hundreds of small decisions each programmer makes along the way. Inevitably, similar pieces of code end up doing similar things in slightly different ways and do not work as well together as they should.

Class libraries provide lots of flexibility. As programs grow more complex, more programmers are forced to reinvent basic solutions to basic problems over and over again. A relatively new extension of the class library idea is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus, windows, dialog boxes, and other standard user interface elements for personal computers.

Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the

flow of control. This was appropriate for printing out paychecks, calculating a mathematical table, or solving other problems with a program that executed in just one way.

The development of graphical user interfaces began to turn this procedural programming arrangement inside out. These interfaces allow the user, rather than program logic, to drive the program and decide when certain actions should be performed. Today, most personal computer software accomplishes this by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code according to actions that the user performs. The programmer no longer determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing control in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

Even event loop programs require programmers to write a lot of code that should not need to be written separately for every application. The concept of an application framework carries the event loop concept further. Instead of dealing with all the nuts and bolts of constructing basic menus, windows, and dialog boxes and then making these things all work together, programmers using application frameworks start with working application code and basic user interface elements in place. Subsequently, they build from there by replacing some of the generic capabilities of the framework with the specific capabilities of the intended application.

Application frameworks reduce the total amount of code that a programmer has to write from scratch. However, because the framework is really a generic application that displays such as windows, supports copy and paste, and so on, the programmer can also relinquish control to a greater degree than event loop programs permit. The framework code takes care of almost all event handling and flow of control, and the programmer's code is called only when the framework needs it (e.g., to create or manipulate a proprietary data structure).

A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

A framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

- Behavior versus protocol. Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.
- Call versus override. With a class library, the code the programmer writes instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together.
- Implementation versus design. With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user

interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved.

A more detailed description of OOP is given by Cotter, et al, Inside Taligent Technology, Addison-Wesley Publishing (1995).

Incorporation of the principles of OOP in a messaging interface allows the messaging system to be more tightly integrated with other OOP-based applications and operating systems. In addition, the maintenance and development effort required by such an OOP-based messaging interface will likely be significantly less than complex procedural programming-based messaging interfaces. This is because parts (i.e., objects or classes) of the OOP-based messaging interface may be modified as needed and automatically propagated throughout the software code without affecting the rest of the messaging interface. In contrast, an entire procedural programming-based messaging interface, as conventionally produced, must be completely tested and debugged each time any modification is made to the software code, because each modification must be manually propagated to different parts of the software code.

The present invention provides a solution to these and other problems, and offers other advantages over the prior art.

Summary of the Invention

The present invention relates to an OOP-based messaging interface between a client application and an electronic messaging system.

In accordance with one aspect of the invention, a messaging interface between a client application and a messaging service for a computer system is provided. The messaging interface includes a storage device, in the computer system, which stores OOP-based classes. These stored classes include: a message class which has data members and member functions related to a message in the messaging service, a

message bin class which has data members and member functions related to holding the message, and a message bin name class which has data members and member functions related to identifying a message bin object instantiated from the message bin class. The messaging interface also includes a processor operatively coupled to the storage device which generates a messaging interface object for the message as an instance of one or more of the stored classes. This generated messaging interface object furnishes the client application with protocol independent access to the messaging service.

This aspect of the invention also can be implemented as an OOP-based messaging interface method for use between a client application and a messaging service on a computer system. This method is performed by device-implemented steps in a series of distinct processing steps that can be implemented in one or more processors. A message class which has data members and member functions related to a message in the messaging service is provided. Also, a message bin class which has data members and member functions related to holding the message is provided. In addition, a message bin name class which has data members and member functions related to identifying a message bin object instantiated from the message bin class is provided. A messaging interface object for the message is generated as an instance of one or more of the provided classes. This messaging interface object furnishes the client application with protocol independent access to the messaging service.

These and various other features as well as advantages which characterize the present invention will be apparent upon reading of the following detailed description and review of the associated drawings.

Brief Description of the Drawings

FIG. 1 is a block diagram of a personal computer system in accordance with a preferred embodiment of the present invention.

FIG. 2 is a block diagram of a route a message might take from an originator to a recipient.

FIG. 3 is a block diagram showing the transport of data through a messaging system.

FIG. 4 is a block diagram of a typical messaging system.

FIG. 5 is a block diagram of components of a preferred embodiment messaging system architecture.

FIG. 6 is a block diagram of a preferred embodiment object oriented programming based messaging interface used in the computer system shown in FIG. 1.

FIG. 7 is a block diagram of the class hierarchy for the preferred embodiment messaging interface.

FIG. 8 is a block diagram of an example life-cycle of a message in the messaging system.

FIG. 9 is a block diagram of the class hierarchy for the preferred embodiment message bin classes.

FIG. 10 is a block diagram of the class hierarchy for the preferred embodiment message bin name classes.

FIG. 11 is a block diagram of the class hierarchy for the preferred embodiment message classes.

FIG. 12 is a flowchart of the preferred embodiment object oriented programming based messaging interface method used in the computer system shown in FIG. 1.

Detailed Description

The preferred embodiment of the present invention is preferably practiced in the context of an operating system resident on a personal computer such as the IBM® PS/2® or Apple® Macintosh® computer. A representative hardware environment is depicted in FIG. 1, which illustrates a typical hardware configuration of a workstation in accordance with the preferred embodiment having a central processing unit **110**, such as a microprocessor, and a number of other units interconnected via a system bus **112**. The workstation shown in FIG. 1 includes a Random Access Memory (RAM) **114**, Read Only Memory (ROM) **116**, an I/O adapter **118** for connecting peripheral devices such as disk storage units **120** to the bus **112**, a user interface adapter **122** for connecting a keyboard **124**, a mouse **126**, a speaker **128**, a microphone **132**, and/or other user interface devices such as a touch screen (not shown) to the bus **112**, communication adapter **134** for connecting the workstation to a communication network (e.g., a data processing network) and a display adapter **136** for connecting the bus **112** to a display device **138**. The workstation typically has resident thereon an

operating system such as the IBM OS/2® operating system or the Apple System/7® operating system.

Store and forward messaging is a class of communication involving asynchronous transmission and reception of data. Asynchronous communication allows message transfer even when the originator and the recipient are not simultaneously active. The term "messaging" sometimes will be used hereinafter for store and forward messaging.

FIG. 2 is a block diagram showing a route that a message might take from an originator 200 to a recipient 204, 208. This route is not usually specified in the message, but is determined by the messaging system. The originator 200 creates the message and submits it to the messaging system. The message is made persistent locally. The message is then forwarded to the next hop and made persistent there at the intermediate node 206. This processes of persisting and forwarding is repeated between the intermediate node 206 and recipient2 208. The message is received by the recipient. At this time, the message may or may not persist after this point; however, usually it does persist. A copy of the message may similarly be delivered to Recipient1 204 through some intermediate nodes 202.

It is important to note that messages persist (i.e., are stored), and that they may be forwarded several times before reaching their destination. If any of the intermediate links are down, the message is held at that hop until the next hop can be reached. This is a basic description of store and forward messaging, including:

Store and forward messaging has many advantages:

- All of the nodes along the route do not have to be up simultaneously.
- The originator does not need to be directly connected to the recipient.
- Timing is less restricted, so translation at intermediate nodes is more feasible, allowing for interconnection of different messaging systems.
- Messages are held in a persistent state and are therefore more immune to failures.

Store and forward messaging also has the following drawbacks:

- Failures can occur asynchronously with respect to submission context forcing more complicated error reporting and status interrogation machinery.

- Timing is unpredictable.
- Gateways and translation may result in loss of data fidelity.
- Reception is not guaranteed.
- Error reporting generally uses messaging facilities as well, so error notifications may be lost.
- Retransmission unit is usually the whole message, which can be quite large.
- Intermediate hops must have persistent storage available.
- Messages are handled by entities who are not the originator and recipient, opening a potential security hole in the system.
- Since the connectivity is not direct, originators may need help in finding recipients, either by being explicitly told of their existence or by browsing some directory.

In general, messages can carry any data. In practice, as shown in FIG. 3, many messaging systems are restricted to carrying a text stream from a limited character set. This is a consequence of the fact that many messaging systems had their roots in facilities designed to provide electronic mail between human users. Electronic mail is still one of the main clients of messaging, and many messaging services are in fact implemented on the remnants of old e-mail systems.

As a result of this, messages containing non-text data may need to be encoded when sent using older messaging systems. In addition, the capacities of these older mail systems are based on 1960's and 1970's technology, which may limit the maximum message size (i.e., messages larger than this size may be truncated, dropped or bounced back).

Because of the ability to gateway systems together, the picture of the messaging systems of today and the near future may be even more complicated. It is possible to join heterogeneous systems in many different combinations. This is very common in today's business environments. For example, FIG. 4 shows a typical messaging system. At least three different messaging systems are depicted here. Some are directly connected, meaning that some form of gateway exists between them and that at least some messages are capable of being routed between the different systems. An example of this is QuickMail 400 and Internet 402. Others are indirectly connected, meaning that they are directly connected to the same third system. An example is

Compuserve 404 and QuickMail 400 being indirectly connected through Internet 402.

Several important points to remember about heterogeneous environments are:

- Connectivity (direct and especially indirect) of different messaging systems may result in loss of data fidelity due to translations and transformations.
- There may be several ways to address a recipient.
- A single workstation may participate in multiple messaging systems.
- Different systems offer different services and guarantees about those services. Requests made of one system may not be honored by another. This can happen without the originator's knowledge or permission.

The present invention provides an interface or a front end to such a messaging system. The interface presented in the following sections attempts to provide choices to clients to ensure the level of integrity and interoperability desired. The interface achieves the following goals:

- The interface is protocol independent (e.g., it is not specifically designed for use with only the Internet or some other electronic mail system).
- The interface supports any content type.
- The interface supports property-based message filtering.
- The underlying framework supports multiple messaging protocols.
- The underlying framework allows implementations of multiple messaging protocols to be concurrently active.

FIG. 5 shows the components of a preferred embodiment messaging system architecture. The messaging interface 500 is a set of classes that client applications (e.g., E-mail 502 and Groupware Applications 504) use to access messaging features. The messaging framework 506 is a set of classes which provide the functionality specified by the messaging interface 500. A service provider plugs into the framework 506 and provides an implementation for functionality that is specific to a messaging protocol or a storage structure. Three service providers are shown, including: Simple Mail Transfer Protocol (SMTP) 508, cc:mail 510, and X.400 512.

The messaging interface 500 communicates between a client application (e.g., E-mail 402) and a messaging service for a computer system (e.g., a messaging framework 406). An example of how this messaging interface 500 might be implemented is an OOP-based environment is shown in FIG. 6. A storage device (e.g., RAM 114, ROM 116, and/or hard disk 120) stores object oriented programming based classes. The basic functions which need to be embodied in classes are data members and member functions related to: a message in the messaging service, a bin for holding the message, and mechanism for identifying a message bin. These functions can be stored in one or more classes. In this example, each of these functions is in a different class (i.e., class 1, class 2, and class 3). A processor 110 retrieves classes from the storage device 114 and generates a messaging interface object for the message as an instance of the retrieved class. This generated messaging interface object alone or in conjunction with other generated objects furnishes the client application with protocol independent access to the messaging service. By using high-level objects, communications from the client applications can be generalized for any service provider or communications protocol. These generalized communications are then interpreted by the underlying messaging framework and converted to a compatible format before sending them to the service provider (e.g., an Internet service provider). Similar conversions are done by the messaging framework through the messaging interface in the receiving direction so that both directions of communication are seamless to the client applications.

The following description is focused on the messaging interface classes and general design principles used in applying them.

FIG. 7 shows the class hierarchy for the preferred embodiment messaging interface. The class tree consists of many abstract classes needed to provide clear demarcation of behaviors and concrete classes that client applications can directly use. The concrete classes are shown with bold and italic text in this and the following figures.

Appropriate interface classes are multi-thread safe for client application's usage, but some classes are not. Later sections will describe each class in detail and specify whether or not it has thread-safety.

Issues of ownership of a heap-allocated object can be quite problematic. The messaging interface classes are generally reference-counted, and manipulation of objects is through the safe-pointer objects

to memory locations. This design frees messaging client applications from the responsibility of storage-deallocation and thus is considered memory safe.

The messaging interface has many classes that act on behalf of a real object -- in some cases persistent objects (e.g., messages, message boxes, etc.). The later sections will identify these classes as surrogate classes. The classes that stand for themselves are labeled value-like. One important difference a client application might notice between surrogate and value-like classes is that an operation on a surrogate object may fail due to the demise of the real object backing it. For example, the command "get subject" operation on a message object will fail, if the message object instance it is referring to has been deleted.

The objects in TMessage hierarchy are thin handles with hidden master objects behind them. Hence these objects are cheap to copy.

Different classes in TMessage 724 hierarchy represent a message at different stage in its life-cycle through the messaging system. TDraftMessage 726 part of the tree represents the messages in construction. These messages are not yet submitted for the delivery and allow adding data and messaging attributes to them. TMessageReference 712 on the other hand represent a message that has been either submitted or received and hence immutable. FIG. 8 shows the life-cycle of a message. Also, since message classes other than TMessageReference 712 are used for construction only, they can not be copied. This means that only one copy of a particular draft message exists at a time thus avoiding conflicting access to same message.

The basic concept and behavior for all the interface classes of the preferred embodiment are described in the following section. A simple basic attributes table is provided for every class. This table provides a quick way to find out about many basic attributes of a class. A description of these basic attributes is given in Table 1 below.

Table 1

Multi-thread-safe	An instance of this class can be concurrently accessed by more than one thread.
Multi-instance-safe	Multiple instances of this surrogate class can be concurrently used to access the real object.
Static-safe	It is OK to construct an instance of this class at static construction time.

Shared-memory-safe	It is OK to keep an instance of this class in shared-memory.
Surrogate Value	Describes whether an instance of this class is a surrogate for some other object.
Allows inheritance	An instance of this class can be subclassed. Some classes are designed to be used monomorphically.
Allocation	Describes whether this class allows creation of instances on the stack or heap or both.
Abstract/Concrete	Describes whether this class allows creation of objects.
Scope of uniqueness	Provided only for identifier classes, it defines the name-space within which the identifier is unique.

Incidental system functions like assignment operator, copy constructor, and streaming operators.

The messaging system needs to keep messages in some storage device (e.g., a magnetic or optical disk drive, a RAM, or a floppy disk). message bin classes provide abstractions for different kinds of message storage. A message is always associated with a message bin. FIG. 9 is a block diagram of the class hierarchy for the preferred embodiment message bin classes.

The root of the hierarchy is an abstract mixin class, message bin class 700, which represents a message bin of any kind. An example of this class is implemented in the preferred embodiment as the MMessageBin class 700 and could be implemented in any other generic message bin class. There are two main kinds of bins. First, there are bins that allow storing messages into them. Second, there are bins that allow retrieving messages from them. The former is represented by the message store mixin class 702 and the latter is represented by the message source mixin class 704. An example of these classes are implemented in the preferred embodiment as the MMessageStore class 702 and MMessageSource class 704. Each of these could be implemented in any other generic message store class and message source class, respectively. Concrete subclasses that allow both storage and retrieval of messages will inherit from both mixins. The preferred embodiment messaging system provides such concrete classes.

The message sender class provides the functionality for message submission and sending. An example of this class is implemented in the preferred embodiment as the MMessageSender class 706 and could be implemented in any other generic message sender class. A sender always needs a store to keep the submitted messages. In the preferred

embodiment, to simplify client application's usage, a MMessageSender 706 is a MMessageStore 702 as well. Similarly, receiver functionality is provided by message receiver class 708 for receiving messages at a receiver-address. An example of this class is implemented in the preferred embodiment as the MMessageReceiver class 708 and could be implemented in any other generic message receiver class. It also is a MMessageSource 704 so that client applications can retrieve messages directly from it.

The MMessageBin class 700 is the root of the bin class hierarchy. It defines a protocol that all bin classes must support as noted in Table 2 below.

Table 2

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
MReferenceCounted

Constants, Enumerators, Types:
- enum
  EStatus {kAvailable, kUnavailable}

Member Functions:
- virtual EStatus
  GetStatus () = 0
- virtual bool
  Contains (const TMessageReference& message)
- virtual bool
  DeleteAll () = 0
- virtual TCountedPointerTo<TMessageBinName>
  GetName () = 0

The enumerator Estatus reflects the status of a bin. In particular a bin may be currently unavailable (e.g., the network connection may be broken). A message bin will attempt reconnection upon a function-call if it is unavailable. One status is "kAvailable" which means that a message bin is currently accessible. Another status is "kUnavailable" which means that a message bin is currently inaccessible.

The GetStatus function returns the status of this bin. If the returned value is kUnavailable, the bin is currently inaccessible. If the

bin is unavailable and trvReconnectIfUnavailable is true, this call will attempt reconnection before returning the status.

The Contains function returns a true value, if the specified message itself is contained within the storage represented by the bin.

The DeleteAll function deletes all messages in this bin. After successful completion, this function will leave the bin empty. A successful completion is indicated by the returned value of true. This function will not delete a message that is currently being accessed. In that case, it returns a false value after deleting all other messages.

The GetName function returns the message bin name for this bin.

The MMessageSource class 704 is a subclass of MMessageBin 700 that adds message retrieval protocol as noted in Table 3 below.

Table 3

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
MMessageBin

Member Functions:

- virtual TMessageReference GetAllMessages (TCollectionOf<TMessageReference> & returnedMessages) const = 0
- virtual unsigned long GetMessageCount() const=0
- virtual TMessageReference WaitForMessageAfter (const TMessageReference& startingPoint) = 0
- virtual TOnlyPointerTo<TMessageIterator> CreateIterator () const

The GetAllMessages function adds the message references of all the messages in this bin to the supplied collection. Previous contents of the collection are left unchanged.

The GetMessageCount function returns the number of messages contained in this bin.

The WaitForMessageAfter function blocks the calling thread until this bin has a message whose LocalCreateTime is later than the LocalCreateTime of startingPoint. If an invalid TMessageReference is passed then the starting LocalCreateTime is assumed to be

TTime::kNegativeInfinity, and the first message in the bin will be returned.

The CreateIterator creates an instance of the message iterator class 710 to iterate over the messages in this bin. An example of this class is implemented in the preferred embodiment as the TMessageIterator class 710 and could be implemented in any other generic message iterator class.

The TMessageIterator class 710 provides iteration functionality over a collection of messages, primarily a message source as noted in Table 4 below.

Table 4

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Value-like
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
None

Member Functions:

- virtual TMessageReference First () = 0
- virtual TMessageReference Next () = 0
- virtual TMessageReference GetMostRecent () const = 0

The First function returns the first message in this bin. If the bin is empty, an invalid TMessageReference object is returned.

The Next function returns the next message in this bin. If the bin is empty, an invalid TMessageReference object is returned.

The GetMostRecent function returns the most recent TMessageReference returned by either the First or Next function. If the bin is empty or no calls to First or Next function have been made, an invalid TMessageReference object is returned.

The MMessageReceiver class 708 is a subclass of MMessageSource class 704 that can receive messages and adds protocols to get its address as noted in Table 5 below.

Table 5

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
MMessageSource

Member Functions:

- virtual TCountedPointerTo<TMessageReceiver Address>
  GetName () = 0

The GetAddress function returns the address for this receiver. It may be used to send messages to this receiver.

The MMessageStore class 702 is a subclass of MMessageBin class 700 and adds message storing protocol as noted in Table 6 below.

Table 6

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
MMessageBin

Member Functions:

- virtual TMessageReference
  CopyInto (const TMessageReference& messageReference,
  bool allowDuplicates = false)

The CopyInto function copies the specified message into this bin. If the message is contained in this bin, another copy is made if allowDuplicates is true. It returns a reference to the new message. If the message is contained in this bin and allowDuplicates is false, then an invalid reference is returned.

The MMessageSender class 706 is a subclass of MMessageStore class 702 and adds protocol for sending messages as noted in Table 7 below.

Table 7

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
MMessageStore

Member Functions:

- virtual TMessageReference Submit (const TDraftMessage& message) = 0
- static TCountedPointerTo<MMessageSender> GetDefaultSender () = 0

The Submit function submits a message in draft state for delivery to its recipients. After this call, the message is no longer a draft message. The draft message, passed in as a parameter, is invalidated and a message reference object instantiated from a message reference class **712** which points to the submitted message is returned. An example of this class is implemented in the preferred embodiment as the TMessageReference class **712** and could be implemented in any other generic message reference class.

The GetDefaultSender function returns the default MMessageSender. It is primarily used to submit messages.

The file system message store class **714** is a concrete subclass of MMessageSender class **706** (which is a subclass of MMessageStore class **702**) and a subclass of MMessageSource class **704**. It represents a file-system-based message bin. Messages are stored in a file system directory. It provides functionality for storing, accessing and sending messages as noted in Table 8 below. An example of this class is implemented in the preferred embodiment as the TFileSystemMessageStore class **714** and could be implemented in any other generic file system message store class.

Table 8

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Concrete

Inherits From:
MMessageSender
MMessageSource

Member Functions:
TDirectory
GetDirectory () const
bool
DeleteSelf ()

The GetDirectory function returns the directory used for message storage.

bool

The DeleteSelf function deletes all messages in this bin and, if successful, deletes the bin itself. A successful completion is indicated by the returned value of true. This function will not delete a message that is currently being accessed — in that case it returns false after deleting all other messages and the bin remains intact. If successful, subsequent calls to GetStatus will return kUnavailable — all other calls will result in a TMessageBinUnavailable exception being thrown.

Accessing a specific instance of a message bin requires that the client be able to specify that bin. This is accomplished using a bin name. Once a bin name is obtained or constructed, a bin can be obtained using the bin name. Bin names are also used to address messages.

The message bin name class **716** and all subclasses, shown in FIG. 10, must be displayable everywhere. An example of this class is implemented in the preferred embodiment as the TMessageBinName class **716** and could be implemented in any other generic message bin name class. The protocol for display name is handled by the base class. In addition, they must be transportable everywhere. The default storage and streaming behavior is managed by the base class to ensure that slicing never occurs.

An instance of the TMessageBinName class **716** identifies a message bin. Its primary purpose is to allow a client to access a message bin. A TMessageBinName class **716** supports the GetPresentationName and GetTextPresentation functions. The TMessageBinName class **716**

does not slice so that all of the data of the original class as well as the type is preserved as noted in Table 9 below.

Table 9

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Value-like
Multi-instance-safe	n/a	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Concrete
		Scope of uniqueness	Global

Inherits From:
MReferenceCounted

Member Functions:

- TCountedPointerTo<MMessageBin>
  GetBin () const
- void
  GetPresentationName (TText& fillin) const
- void
  GetTextRepresentation (TText& fillinByAppending) const

The GetBin function returns a counted pointer to the message bin. If the TMessageBinName class **716** is invalid, or the class library for the specified bin is not available, an exception is thrown.

The GetPresentationName function returns the presentation name of the message bin. The presentation name is that which would be used when displaying the name of the bin to the user.

The GetTextRepresentation function returns a text representation of the message bin name. Many message bins have a valid text representation for the bin name. This text representation is usually sufficient to fully specify the bin.

An instance of the file system message store name class **714** identifies a file system message store as noted in Table 10 below. An example of this class is implemented in the preferred embodiment as the TFileSystemMessageStoreName class **714** and could be implemented in any other generic file system message store name class.

Table 10

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Value-like
Multi-instance-safe	n/a	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Concrete
		Scope of uniqueness	Global

Inherits From:
TMessageBinName

Member Functions:
TFileSystemMessageStoreName (const TDirectory& directory)

- TCountedPointerTo<TFileSystemMessageStore>
  GetStore () const

- TDirectory
  GetDirectory () const

The TFileSystemMessageStoreName function constructs a file system message store over a directory. This call throws an exception if the directory is invalid.

The GetStore function returns a counted pointer to a file system store.

The GetDirectory function returns a directory used for messages storage.

An instance of the TMessageReceiverAddress (also known as MessageReceiverAddress) class 718 identifies a message receiver. Its primary purpose is to allow a client to access a message receiver and for constructing a message address bundles 720 from its class for sending messages as noted in Table 11 below. An example of this class is implemented in the preferred embodiment as the TMessageAddressBundle class 720 and could be implemented in any other generic message address bundle class.

Table 11

Basic Attributes:

Multi-thread-safe	Yes	Surrogate/Value-like	Value-like
Multi-instance-safe	n/a	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Concrete
		Scope of uniqueness	Global

Inherits From:
TMessageBinName

Member Functions:

- virtual TCountedPointerTo<MMessageReceiver> GetMessageReceiver () const

The GetMessageReceiver function returns a counted pointer to a message receiver. It throws an exception, if the receiver cannot be constructed, or if an instance of TMessageReceiverAddress class 718 is invalid.

Particular service provider classes can be subclassed from any of these previously described classes to further enhance the functionality of the messaging interface 400. For example, a Internet receiver address class 722 can be generated by inheriting functionality from the TMessageReceiverAddress class 718 to add Internet addressing capabilities.. An example of this class is implemented in the preferred embodiment as the TInternetReceiverAddress class 722 and could be implemented in any other generic internet receiver address class. Similarly, classes can be generated for other messaging systems such as X.400.

An instance of the TInternetReceiverAddress class 722 identifies an RFC-822 compliant mailbox address as noted in Table 12 below.

Table 12

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Value-like
Multi-instance-safe	n/a	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Concrete
		Scope of uniqueness	Global

Inherits From:
TMessageReceiverAddress

Member Functions:

TInternetReceiverAddress (const TText& RFC-822address)

TInternetReceiverAddress (const TText& userName, const TText& hostName, const TText& presentationName = TStandardText::GetEmptyText())

- void GetParts (TText& localPart,TText& domainPart) const

The TInternetReceiverAddress function constructs a TInternetReceiverAddress from an RFC-822 conformant string. This will throw an exception, if the passed string is not a valid RFC-822 address.

The TInternetReceiverAddress function constructs an Internet address from components that can be used to form a RFC-822 conformant string.

The GetParts function returns local and domain parts of the RFC-822 address. For a more detailed discussion of the Internet messaging standard and the syntax of an RFC-822 Internet address, see Request For Comments 822 (RFC-822) by Dave Crocker dated August 13, 1982.

A message class 724 represents a message. An example of this class is implemented in the preferred embodiment as the TMessage class 724 and could be implemented in any other generic message class. There are many kinds of messages in a messaging system. For example, some kinds are messages received at an address, messages sent from an address, and messages in construction yet to be sent.

Various subclasses of the TMessage class 724 represent different kinds of messages as shown in FIG. 11. The TMessage class 724 has the basic functionality for all of them as noted in Table 13.

Table 13

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Value-like
Multi-instance-safe	n/a	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Heap
Shared-memory-safe	No	Abstract/Concrete	Concrete

Inherits From:
None

Constants, Enumerators, Types:

- ComponentIndex kInvalidIndex
  An invalid ComponentIndex.
- enum
  EStatus {kDraft, kPending, kSending,
  kSent, kReceived, kFailed}
- enum
  EImportance {kBulk, kNormal, kImportant}
- enum
  EStreamFormat {kMonomorphic, kPolyMorphic}

Member Functions:

- bool
  IsValid () const
- bool
  operator== (const TMessage& message) const
- bool
  operator!= (const TMessage& message) const
- bool
  operator > (const TMessage& message) const
- bool
  operator < (const TMessage& message) const
- bool
  operator >= (const TMessage& message) const
- bool
  operator <= (const TMessage& message) const
- HashResult
  Hash () const
- bool
  IsContainedIn (const MMessageBin& messageBin) const
- unsigned long
  CountAddresses (TMessageAddressBundle::KindSet filter) const
- ComponentIndex
  CountComponents () const
- ComponentIndex
  DoesContainType (const TTypeSurrogate& componentType) const
- bool
  IsComponentType (const TTypeSurrogate& componentType,
  ComponentIndex index) const
- void
  GetComponentType (TTypeSurrogate& fillin,
  ComponentIndex index) const
- EStreamFormat
  GetComponentStreamFormat (ComponentIndex index) const
- void
  GetSubject (TText& fillin) const
- EStatus
  GetStatus () const
- EImportance
  GetImportance () const

- TTime
  GetCreateTime () const
- TTime
  GetLocalCreateTime () const
- TCountedPointerTo<MMessageBin>
  GetBin () const
- void
  DeleteSelf () const

The EStatus enumerator reflects the status of a message. The message may have a status of many different types. For example, "kDraft" means that an instance of the TMessage class 724 has not yet been submitted for delivery. All newly created messages are in this state and remain in this state until the client application calls Submit or Send on it. Another status is "kPending" which means that an instance of the TMessage class 724 was submitted by the client application for delivery but the messaging system has not yet sent it to a messaging server. Another status is "kSending" which means that an instance of the TMessage class 724 is currently being sent to a messaging server. Another status is "kSent" which means that an instance of the TMessage class 724 was successfully sent to a messaging server. Another status is "kReceived" which means that an instance of the TMessage class 724 was received by the messaging system. Another status is "kFailed" which means that an instance of the TMessage class 724 could not be delivered.

The EImportance enumerator has many values, including: "kBulk", "kNormal", and "kmportant". Each is an indication of the client application's assessment of the importance of this message. "kBulk" is the least important and "kImportant" is the most. This tag (enumerator) is mutually interpreted by the messaging client applications, because it is not interpreted by the messaging system.

The EStreamFormat enumerator preferably has two values. One value is "kMonomorphic", which indicates that an instance of the TMessage class 724 was added using the AppendMonomorphic function, and must be retrieved using GetMonomorpic function. Another value is "kPolymorphic", which indicates that an instance of the TMessage class 724 was added using the AppendPolymorphic function, and must be retrieved using the GetPolymorphic function.

The IsValid function checks if an instance of this class is a valid message. A message created using an empty constructor and a message

reference returned at the end of an iteration are invalid references. It returns a true, if the message is valid. Most operations on an invalid message will fail. Operations on a valid reference might still fail, if the message has been deleted or otherwise unavailable for access.

The operator== and operator!= functions check for equality and inequality, respectively, between two messages. The first returns a true and the second returns a false, respectively, only if both references refer to the same message. The other operator functions perform an ordered comparison check for two messages. Message order is based on the LocalCreateTime.

The Hash function returns a hash key for this message.

The IsContainedIn function returns a true if this message is from the storage represented by messageBin.

The CountAddress function returns the number of address in this message whose kind matches filter.

The CountComponents function returns the number of components in this message.. The DoesContainType function returns the index of the first component of type described by componentType. It returns kInvaldIndex, if no matching type is found. The IsComponentType function returns a true if the component at specified index is of type described by componentType. The GetComponentType function returns the type of component at specified index in fillin. The GetComponentStreamFormat function returns the streaming format for component at specified index in fillin.

The GetSubject function assigns the subject for this message to fillin. The GetStatus function returns the status of this message. The GetImportance function returns the importance tag for this message. The GetCreateTime function returns the time in UTC (Universal Coordinated Time) when this message was created. The GetLocalCreateTime function returns the time in UTC (Universal Coordinated Time) when this message was created in the current bin. Will == create time unless this message was copied from another bin (see MMessageStore::CopyInto) in which case it is guaranteed to be >= create time. The GetBin function returns a counted pointer to the message bin that contains this message.

The DeleteSelf function deletes the message referenced by this object. The DeleteSelf function may fail if the message is a draft

message and is currently being accessed. Failure typically is indicated by an exception.

The TMessageReference class 712 is a subclass of the TMessage class 724. It represents either a received or submitted message. A message of this kind can not be modified as noted in Table 14 below.

Table 14

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Bith
Shared-memory-safe	No	Abstract/Concrete	Concrete

Inherits From:
TMessage

Constants, Enumerators, Types:

- enum
  EIntegrity {kIntact, kDoNotKnow}

Member Functions:

- TTime
  GetSubmitTime () const
- TTime
  GetReceiveTime () const
- EIntegrity
  GetIntegrity () const
- TSTream&
  operator>>= (TStream& toWhere) const
- TSTream&
  operator<<= (TStream& fromWhere)

The enumerator EIntegrity can have a "kIntact" value, which means that an instance of the TMessage class 724 was received without any loss of data. Alternatively, a "kDoNotKnow" status means that an instance of the TMessage class 724 does not have enough information to judge its integrity.

The GetSubmitTime function returns the time in UTC (Universal Coordinated Time) when this message was submitted for delivery. The GetReceiveTime function returns the time in UTC when this message was received. If the message status is not Kreived, then the message has never been received (probably because it is an outgoing message) and the time returned is TTime::kPositiveInfinity.

The GetIntegrity function returns the integrity with which this message was received.

The operator>>= and operator<<= functions streams this message reference and not the message data to and from, respectively, a stream.

The TMessage class 724 is generally useful for referring to any kind of a message and accessing the message data. It does not provide a protocol for the creation and addition of data, because not all messages support that behavior. For example, received messages are immutable. The draft message class 726 and its subclasses provide functionality for constructing a new message, writing data to it and sending it. An example of this class is implemented in the preferred embodiment as the TDraftMessage class 726 and could be implemented in any other generic draft message class.

A message sent from an OOP-based messaging system may travel through a series of messaging routers (most of which will be non-OOP-based for quite some time). A recipient for the message may be a non-OOP-based messaging system. In such an environment, the issues of integrity of a message through the delivery and its interoperability with non-OOP-based systems needs to be clearly defined. Several subclasses of the TDraftMessage class 726 may be defined to support different trade-offs between data-integrity and interoperability.

For example, the OOP-specific message class 728 guarantees the data-integrity when delivered to a recipient. An example of this class is implemented in the preferred embodiment as the TCommonPointMessage class 728 and could be implemented in any other generic message class. However, only an OOP-based messaging system (e.g., a CommonPoint™ messaging system) will be able to retrieve the data from an instance of the TCommonPointMessage class 728. CommonPoint™ is a trademark of Taligent, Inc. In contrast, the interoperable message class 730 is interpretable by and can be sent to a non-OOP-based messaging system. It, however, does not guarantee data integrity. In particular, an instance of the interoperable message class 730 may have undergone lossy transformation of message contents. An example of this class is implemented in the preferred embodiment as the TInterOperableMessage class 730 and could be implemented in any other generic interoperable message class.

A CommonPoint™ messaging system client application will choose a kind of draft message based on the requirements for data

integrity and interoperability dictated by the problem domain as noted in Table 15.

Table 15

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Both
Shared-memory-safe	No	Abstract/Concrete	Abstract

Inherits From:
TMessageReference

Constants, Enumerators, Types:

- enum
  EReplyRecipientHandling
  {kReplyToSenderOnly,kReplyToAllRecipients}
- enum
  ECopyRecipientHandling {kCopyNoRecipients,kCopyAllRecipients}
- enum
  ESubjectHandling {kDoNotCopySubject, kCopySubject}
- enum
  EComponentHandling {kCopyNoComponents, kCopyAllComponents}

Member Functions:

- void
  AppendAddress (const TMessageAddressBundle& bundle)
- void
  AppendAddress (const TCountedPointerTo<TMessageReceiverAddress>&
  address, TMessageAddressBundle::EKind = TMessageAddressBundle::kTo)
- void
  AppendComponent (const TMessageReference& fromWhere,
  ComponentIndex index)

The following enumerators control message creation behavior, when a draft message is created as a reply, forward, or copy of an existing message.

The enumerator EReplyRecipientHandling controls recipient handling when creating a draft message as a reply o a message. It may have a "kReplyToSenderOnly" value which uses the sender of the original message as a recipient. Alternatively, it may have a "kReplyToAllRecipients" value which uses all of the recipients of the original message as recipients.

The enumerator ECopyRecipientHandling controls recipient handling when creating a draft message as a copy of a message. It may have a "kCopyNoRecipients" value which does not copy any recipients from the existing message. Alternatively, it may have a

"kCopyAllRecipients" value which copies all recipients from the existing message.

The enumerator ESubjectHandling controls the subject handling when creating a draft message as a forward, reply or copy of a message. It may have a "kDoNotCopySubject" which indicates that the subject of the original message is not to be copied to this message. Alternatively, it may have a "kCopySubject" value which indicates that the subject of the original message may be copied to this message.

The enumerator EComponentHandling controls the component handling when creating a draft message as a forward, reply or copy of a message. It may have a "kCopyNoComponents" value which indicates that none of the components of the original message are to be copied to this message. Alternatively, it may have a "kCopyAllComponents" value which indicates that the components of the original message may be copied to this message.

The function AppendAddressBundle adds an address bundle to the list of addresses for this draft message. It may also add an address bundle with the specified address and kind to the list of addresses for this draft message.

The function AppendComponent copies component at index in message fromWhere to next index in this message and returns an address to the list of addresses for this draft message.

The TCommonPointMessage class 728 is a concrete subclass of TDraftMessage class 726. This message supports multiple components of any data-type and styled-text as subject. It does not translate the data from its original form to any other as noted in Table 16.

Table 16

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Both
Shared-memory-safe	No	Abstract/Concrete	Concrete

inherits From:
TDraftMessage

Member Functions:
TCommonPointMessage (const TText& subject)

TCommonPointMessage (const TText& subject, const TCountedPointerTo<TMessageReceiverAddress>& toAddress)

- static TCommonPointMessage Forward (const TMessageReference& sourceMessage, EComponentHandling componentHandling, ESubjectHandling subjectHandling, const TText& prependToSubject = TStandardText::GetEmptyText())
- static TCommonPointMessage Reply (const TMessageReference& sourceMessage, EReplyRecipientHandling recipientHandling, EComponentHandling componentHandling, ESubjectHandling subjectHandling, const TText& prependToSubject = TStandardText::GetEmptyText())
- static TCommonPointMessage Copy (const TMessage& sourceMessage, ECopyRecipientHandling recipientHandling, EComponentHandling componentHandling, ESubjectHandling subjectHandling, const TText& prependToSubject = TStandardText::GetEmptyText())

The function TCommonPointMessage allows client applications to create new CommonPoint™ messages with a specified subject. One of the constructors allows specifying an address to which this message should be sent.

The function Forward creates a CommonPoint™ message as a forward of the specified message. The newly created CommonPoint™ message is properly formatted as a forwarded message. Parameters componentHandling and subjectHandling provide clients a finer control of controlling various properties of the created message as described earlier. If prependToSubject is present, the specified text is prepended to the subject of sourceMessage to create subject for the returned message.

The function Reply creates a simple message as a reply to the specified message. The newly created simple message is properly formatted as a reply message. Parameters are handled in manner similar to Forward function.

The function Copy creates a simple message as a copy of the specified message. Parameters are handled in manner similar to Forward function.

The TInterOperableMessage class 730 is a concrete subclass of the TDraftMessage class 724. This message supports multiple components of any data-type and unstyled-text as subject. Upon transmission, it may translate the data from its original form to one appropriate for passage through the messaging system. Such a translation may incur loss of information from the data as noted in Table 17 below.

Table 17

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Surrogate
Multi-instance-safe	Yes	Allows inheritance	Yes
Static-safe	No	Allocation - stack,heap	Both
Shared-memory-safe	No	Abstract/Concrete	Concrete

Inherits From:
TDraftMessage

Member Functions:
TInterOperableMessage (const TText& subject)

TInterOperableMessage (const TText& subject, const TCountedPointerTo<TMessageReceiverAddress>& toAddress)

TInterOperableMessage (const TText& subject, const TCountedPointerTo<TMessageReceiverAddress> toAddress, const TStandardText& firstComponent)

- static TInterOperableMessage Forward (const TMessageReference& sourceMessage, EComponentHandling componentHandling, ESubjectHandling subjectHandling, const TText& prependToSubject = TStandardText::GetEmptyText())
- static TInterOperableMessage Reply (const TMessageReference& sourceMessage, EReplyRecipientHandling recipientHandling, EComponentHandling componentHandling, ESubjectHandling subjectHandling, const TText& prependToSubject = TStandardText::GetEmptyText())
- static TInterOperableMessage Copy (const TMessage& sourceMessage, ERecipientHandling recipientHandling, ECopyComponentHandling componentHandling, ESubjectHandling subjectHandling, const TText& prependToSubject = TStandardText::GetEmptyText())

The function TInterOperableMessage allow client applications to create new interoperable messages with specified subject. One of the constructors allows specifying an address to which this message should be sent. Another constructor supports the (monomorphic) addition of a single text component. Regardless of which constructor is used so that

client applications can add additional components using the AppendMonomorphic and AppendPolymorphic functions.

The function forward creates a CommonPoint™ message as a forward of the specified message (sourceMessage). The newly created CommonPoint™ message is properly formatted as a forwarded message. Parameters componentHandling and subjectHandling provide client applications a finer control of controlling various properties of the created message as described earlier. If prependToSubject is present, the specified text is prepended to the subject of sourceMessage to create subject for the returned message.

The function Reply creates a simple message as a reply to the specified message. The newly created simple message is properly formatted as a reply message. Parameters are handled in manner similar to Forward function.

The function Copy creates a simple message as a copy of the specified message. Parameters are handled in manner similar to Forward function.

A component inside a TDraftMessage class **726** can be any CommonPoint™ object. Functionality of writing a component to a message and reading one from it is provided in a type-safe way by template functions Add and Get. Conceptually, the components in a message form an indexed collection. Client applications can add components to a message sequentially and access a component using an index. Some of these functions are detailed in Table 18 below.

Table 18

Functions:

- template <class AType> ComponentIndex AppendPolymorphic (TDraftMessage& message, const AType& component)
- template <class AType> ComponentIndex AppendMonomorphic (TDraftMessage& message, const AType& component)
- template <class AType> void GetPolymorphic (const TMessage& message, TOnlyPointerTo<AType>& fillin, ComponentIndex index)
- template <class AType> void GetMonomorphic (const TMessage& message, AType& fillin, ComponentIndex index)

The function AddPolymorphic adds a component to a message at the next available index. The component is placed in the message body by the process of polymorphic streaming (by calling template <class

AType> void Flatten (const AType*,TStream&)). It returns the index at which the component is placed.

The function AddMonomorphic adds a component to a message at the next available index. The component is placed in the message body by the process of monomorphic streaming (by calling AType::operator>>=). It returns the index at which the component is placed.

The function GetPolymorphic retrieves the component in message at index as an object of AType. If the component in message is of different type, this function throws TTypeMismatch exception.

The function GetMonomorphic retrieves the component in message at index as an object of AType. If the component in message is of different type, this function throws TTypeMismatch exception.

Another class is related to message properties. This is the TMessageAddressBundle class which represents an entity to which a message can be sent. Since an address is needed for sending a message, the TMessageAddressBundle class is constructed from a message receiver address and related attributes as noted in Table 19 below.

Table 19

Basic Attributes:

Multi-thread-safe	No	Surrogate / Value-like	Value-like
Multi-instance-safe	Yes	Allows inheritance	No
Static-safe	No	Allocation - stack,heap	Both
Shared-memory-safe	No	Abstract / Concrete	Concrete

Inherits From:
MReferenceCounted

Constants, Enumerators, Types:

- enum
  EKind {kTo, kCC, kBCC, kForwardTo, kFrom, kForwardFrom, kReplyTo, kRecipient, kAll}

Member Functions:

TMessageAddressBundle (const TCountedPointerTo<TMessageReceiverAddress>& address, EKind kind = kTo)

TMessageAddressBundle (TStream& fromWhere)

TMessageAddressBundle ()

- TCountedPointerTO<TMessageReceiverAddress>
  GetAddress () const
- EKind
  GetKind () const
- bool
  IsValid () const

The enumerator EKind reflects an attribute to qualify a receiver address. These can be logically "OR'd" together to form a filter for use in address iteration. One attribute is "kTo" which means a primary target of a message. Another attribute is "kCC" which means a secondary target of a message. In addition, kBCC is a blind secondary target of a message. kForwardTo is a forwarding target of a message. kFrom is the originator of a message. kForwardFrom is the address from which a message was forwarded. kReplyTo is an address used when replying which defaults to kFrom address. kRecipient is a predefined filter for (kTo + kCC + kBCC + kForwardTo). kAll is a predefined filter which will match any kind.

The function TMessageAddressBundle constructs a bundle object with the specified address and kind. Alternatively, it constructs a bundle object by streaming data from the stream fromWhere. Alternatively, it creates an invalid bundle. Such a bundle is useful for future assignment. A null bundle in a draft message is ignored.

The function GetAddress returns the address for this bundle. Also, the function GetKind returns the kind attribute for this bundle. In addition, the function IsValid returns true if the bundle has a valid address.

Another class related to message properties is the TMessageAddressIterator class (not shown) which provides an iterator over the addresses in a message as noted in Table 20 below.

Table 20

Basic Attributes:

Multi-thread-safe	No	Surrogate/Value-like	Value-like
Multi-instance-safe	Yes	Allows inheritance	No
Static-safe	No	Allocation - stack,heap	Both
Shared-memory-safe	No	Abstract/Concrete	Concrete

Inherits From:
None

Member Functions:

TMessageAddressIterator (const TMessage& message,
TMessageAddressBundle::EKindSet filter = kTo)

TMessageAddressIterator ()

- virtual TMessageAddressBundle
First () const = 0

- virtual TMessageAddressBundle
Next () const = 0

- bool
operator== (const TMessageAddressIterator& right) const

- bool
operator!= (const TMessageAddressIterator& right) const

The TMessageAddressIterator function creates an iterator over the specified message with the specified filter. Alternatively is may create an invalid iterator which can be reserved for a future assignment.

The First function returns the first address in this message. If the bin is empty, an invalid TMessageAddressBundle object is returned.

The Next function returns the next message in this bin. If the bin is empty, an invalid TMessageAddressBundle object is returned.

The operator== and operator!= functions are equality (inequality) checks for two iterators. They return a true if both iterators refer to the same message, have the same filter, and have the same iteration 'position'.

Some general items to note are that subclass Application Programming Interface (API) classes are not needed to use the messaging interface classes. The messaging interface API is built on top of a messaging framework. The framework requires subclassing of a set of classes to plug-in a service provider.

The messaging interface API does not guarantee delivery of a message. Different service-providers may have different level of delivery guarantees. Some messages may be delivered with loss of data. The messaging API defines a message type that will not suffer any transformation when delivered to a CommonPoint™ recipient (TCommonPointMessage class 728). There is a type of message which imposes no constraints on the contents or recipients (TInterOperableMessage class 730). However, the messaging interface API makes no guarantees about the integrity of instances of the TInterOperableMessage class 730.

Both TCommonPointMessage class 728 and TInterOperableMessage class 730 support multiple components of any type, as long as the type has the CommonPoint™ type extensions. Instances (i.e., objects) of the TInterOperableMessage class 730 may drop the components that can not be translated appropriately.

By iterating over component types using TMessage::GetComponentType, contents of a message can be determined. In addition, the existence of a specific type of component (say text) can be checked by using TMessage::DoesContainType.

The messaging interface API does not provide any direct means other than construction or reading of received messages for obtaining receiver addresses. The goal is to have receiver address manufactured by other objects such as people or business cards, etc.

Client applications may use service-provider-specific address classes (e.g., the TInternetReceiverAddress class 722) bearing in mind that it results in routing a message to that address through only given service-provider.

To keep the different aspects of storage behavior separate from each other several bin classes are provided. For example, the MMessageStore class 702 provides writing behavior (create or copy a message in to) for a bin, while the MMessageSource class 704 provides reading behavior (iterate over contained messages). This kind of

separation allows for constructing subclasses with various combinations of these behaviors.

This separation of behaviors is strictly enforced. For example, MMessageStore provides write-only protocol to message bin. It would break the write-only behavior to provide wait for or iterate protocol in this class. Thus, it is not possible to wait for (i.e., key off on) a message arrival at a MMessageStore object.

The following example shown in Table 21 illustrates the use of basic functionality provided by the messaging interface.

Table 21

```
// $Revision: 1.2.5
// MessagingSamples.C
// Copyright © 1995 Taligent, Inc. All rights reserved
//
// In order to focus on the Messaging API's, this program uses a traditional
// C style with a main and global functions.

//----------------------------------------------------------------------
// Includes
//----------------------------------------------------------------------

#ifndef Taligent_MESSAGING
#include <Taligent/Messaging.h>
#endif

#ifndef Taligent_INTERNETRECEIVERADDRESS
#include <Taligent/InternetReceiverAddress.h>
#endif

#ifndef Taligent_DOCUMENTSTORECONTENTSTREAMER
#include <Taligent/DocumentStoreContentStreamer.h>
#endif

#ifndef Taligent_SAMPLEPROCEDUREMAPS
#include <Taligent/SampleProcedureMaps.h>
#endif

#ifndef Taligent_EXCEPTIONFORMATTER
#include <Taligent/ExceptionFormatter.h>
#endif

#ifndef Taligent_MULTIPLEROOTLOCATOR
#include <Taligent/MultipleRootLocator.h>
#endif

#ifndef Taligent_BASICDOCUMENTSTORAGE
#include <Taligent/BasicDocumentStorage.h>
#endif

//----------------------------------------------------------------------
// Message Sending Functions
//----------------------------------------------------------------------

// Send a message containing text only. The message can be read within almost
// any mail system on any host. Text styles will be stripped and some loss of
// text may occur due to character set translation.

void SendInteroperableText(const TCountedPointerTo<TMessageReceiverAddress>&
    address,
                           const TStandardText& subject,
                           const TStandardText& text)
{
  TInteroperableMessage message(subject,address,text);  // Make message.
  MMessageSender::GetDefaultSender()->Submit(message);  // Submit.
}

// Send a message containing text only.  The message can only be read within
// CommonPoint™.

void SendCommonPointText(const TCountedPointerTo<TMessageReceiverAddress>& address,
                         const TText& subject,
                         const TStandardText& text)
{
  // Note:  TStandardText objects should normally be handled using
  //        AddMonomorphic and GetMonomorphic functions.  Polymorphic handling
  //        is needlessly expensive.

  TCommonPointMessage message(subject,address);        // Make message
  AppendMonomorphic(message,text);                     // Add the text.
  MMessageSender::GetDefaultSender()->Submit(message); // Submit.
}

// Send a message containing a single CommonPoint™ object. The message can only
// be read within CommonPoint™.

template <class AType>
void SendOneObject(const TCountedPointerTo<TMessageReceiverAddress>& address,
                   const TText& subject,
                   const AType& object)
{
  TCommonPointMessage message(subject,address);        // Make message.
  AppendPolymorphic(message,object);                   // Add the object.
```

```
    MMessageSender::GetDefaultSender()->Submit(message);    // Submit
}

// Send a message containing three CommonPoint™ objects.  The message can only
// be read within CommonPoint™:

template <class AType, class BType, class CType>
void SendThreeObjects (const TCountedPointerTo<TMessageReceiverAddress>& address,
                       const TStandardText& subject,
                       const AType& object1,
                       const BType& object2,
                       const CType& object3)
{
  TCommonPointMessage message(subject,address);        // Make message.
  AppendPolymorphic(message.object1);                  // Add the objects
  AppendPolymorphic(message.object2);
  AppendPolymorphic(message.object3);
  MMessageSender::GetDefaultSender()->Submit(message);    // Submit.
}

// ---------------------------------------------------------------------------
// Message Examination Functions
// ---------------------------------------------------------------------------

// Scan message and return true iff contains a component of the specified type:

bool DoesMessageContain (const TMessageReference& message,
                         const TTypeSurrogate& targetType)
{
  bool found = false;

  for (int i = 0; i < message.CountComponents(); i++)

    if (message.IsComponentType(targetType))

      found = true;
      break;

  return found;
}

// Scan message and print all text components:

void PrintTextComponents (const TMessageReference& message)
{
  TStandardText  text;
  TTypeSurrogate  textType(StaticTypeInfo(TStandardText));
  ComponentIndex  count = message.CountComponents();

  for (int i = 0; i < count; i++)

    if (message.IsComponentType(textType))

      GetMonomorphic(message.text,i); // Extract text from message.
      QPrintText(text);               // Print it.

  }
}

// ---------------------------------------------------------------------------
// Send and Receive Messages
// ---------------------------------------------------------------------------

void MessagingSamples (const TCountedPointerTo<TMessageReceiverAddress>& address,
                       const TText& textSubject,
                       const TStandardText& textBody,
                       const TText& cpSubject,
                       const TdocumentReference& document,
                       const TTime& time,
                       const TProcedureMap& map)
{
    Send an interoperable text message...

  SendInteroperableText(address,textSubject,textBody);

    Send a text object...

  SendCommonPointText(address,textSubject,textBody);

    Send a message containing a single CommonPoint object (a document
    by reference)...
```

```
SendOneObject(address.cpSubject.document);

// Send a message containing a single CommonPoint object (a document
// by value)...

SendOneObject(address.cpSubject.TDocumentStoreContentStreamer(document));

// Send a message containing three CommonPoint objects...

SendThreeObjects(address.cpSubject.document.time.map);

// Wait for the messages to be recieved.  First, get the receiver
// instance...

TCountedPointerTo<MMessageReceiver> receiver = address->GetMessageReceiver();

// Now loop waiting until we recieve the message containing
// a TProcedureMap...

bool          done = false;
TTypeSurrogate mapTypeStaticTypeInfo(TProcedureMap));

TMessageReference receivedMessage;  // Start with an invalid message

while (! done)
{
  // Wait for a new message to arrive...

  receivedMessage = receiver->WaitForMessageAfter(receivedMessage);

  // Print any text components in the message

  PrintTextComponents(receivedMessage);

  // Terminate when we've recieved the  message containing three
  // objects, one of which is a document...

  if (receivedMessage.CountComponents() == 3)
  {
    done = DoesMessageContain(receivedMessage,mapType);
  }
}

// ---------------------------------------------------------------------
// Main
//
// For simplicity, we assume that this program is entered from the command
// line, and that it is passed a single argument which is treated as an
// Internet style mail address.  All other data is hard-coded.
//
// Normally, addresses should be retrieved from either a business card or from
// the directory services.
// ---------------------------------------------------------------------

int main(int argc, char*argv[])
{
  bool    failed = (argc != 2);

  if (! failed)
  {
    TOnlyPointerTo<TDocumentStore>          store;
    TOnlyPointerTo<TStandardStorageMechanism>  mechanism;

    try
    {
      // Make an Internet address...

      TCountedPointerTo<TMessageReceiverAddress> address =
        new TInternetReceiverAddress(TStandardText(argv[1]));

      // Make up the remaining arguments to MessagingSamples...

      TStandardText      textSubject("Text Enclosed.");
      TStandardText      textBody("This is a text message from CommonPoint.");
      TStandardText      cpSubject("CommonPoint Objects Enclosed.");
      TSeconds           time(1234);
      TMarbleProcedureMap map;

      // Get a TDocumentReference...

      TMultipleRootLocator locator("Places");
      TDirectory homePlace = locator.FindFirstByName("DefaultUserHomePlace");
```

```
        mechanism = new TStandardStorageMechanism(homePlace, MessagingTestDoc );
        store = new TDocumentStore(mechanism.Orphan());
        TDocumentReference document = store->GetDocumentReference();

        // Now execute the samples...

        MessagingSamples(address,textSubject,textBody,cpSubject,document,nme,map);
      }

      catch (const TStandardException& exception)
      {
        failed = true;
        TStandardText message;
        TExceptionFormatter::GetFormattedText(exception, TLocale::GetCurrentLocale(),
                                              message);
        QPrintText(message);
      }

      catch (...)
      {
        failed = true;
        QPrintText(TStandardText("An unknown exception was caught."));
      }
    }
    else
    {
      QPrintText(TStandardText("Invalid arguments. Use: MessagingSamples
                               address"));
    }

    return failed ? 1 : 0;
```

The present invention can be summarized in reference to FIG. 12 which is a flowchart of the preferred embodiment object oriented programming based messaging interface method for use between a client application and a messaging service on a computer system. This method is performed by device-implemented steps in a series of distinct processing steps **1200-1212** that can be implemented in one or more processors.

A message class **724** is provided **1202** which has data members and member functions related to a message in the messaging service. In addition, a message bin class **700** is provided **1204** which has data members and member functions related to holding the message. Also, a message bin name class **716** is provided **1206** which has data members and member functions related to identifying a message bin object instantiated from the message bin class **700**. Subsequently, a messaging interface object for the message is generated **1210** as an instance of one of these provided classes (i.e., the message class **724**, the message bin class **700**, or the message bin name class **716**. This messaging interface object provides the client application with protocol independent access to the messaging service.

Other classes may be provided **1208** prior to the generating step **1210** such that other messaging interface-related objects can be generated in the generating step **1210**. For example, a message address bundle class

may be provided 1208 which has data members and member functions related to a message receiver address to which the message can be sent in the messaging service and related attributes of the message receiver address. In addition, a message address iterator class may be provided 1208 which has data members and member functions related to an iterator over the addresses.

A message reference class 712 may be provided 1208 which inherits all of the data members and member functions defined by the message class 724. It further defines data members and member functions related to locking the message such that the message can not be modified.

A draft message class 726 may be provided 1208 which inherits all of the data members and member functions defined by the message class 724. It further defines data members and member functions related to data integrity and interoperability of the message. Further, an OOP-specific message class 728 may be provided 1208 which inherits all of the data members and member functions defined by the draft message class 726. It further defines data members and member functions related to guaranteeing data integrity when delivering the message to an OOP-based message system. Furthermore, an interoperable message class 730 may be provided 1208 which inherits all of the data members and member functions defined by the draft message class 726. It further defines data members and member functions related to guaranteeing data interoperability when delivering the message to a non-OOP-based message system.

A message store class 702 may be provided 1208 which inherits all of the data members and member functions defined by the message bin class 700. It further defines data members and member functions related to protocols for storing the message into a message store object instantiated from the message store class 702. Further, a message sender class 706 may be provided 1208 which inherits all of the data members and member functions defined by the message store class 702. It further defines data members and member functions related to protocols for submitting and sending the message from the client application to the messaging service. In addition, a message source class 704 may be provided 1208 which inherits all of the data members and member functions defined by the message bin class 700. It further defines data members and member functions related to protocols for retrieving the

message from a message source object instantiated from the message source class 704. This allows a file system message store class 714 to be provided 1208 which inherits all of the data members and member functions defined by the message sender class 706 and the MMessageSource class 704. It further defines data members and member functions related to protocols for storing, accessing and sending the message where the message is stored in a file system directory in a storage device 120 of the computer system.

A message source class 704 alone (i.e., without the message sender class 706 and the message source class 704) may be provided 1208 which inherits all of the data members and member functions defined by the message bin class 700. It further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class 704. Further, a message receiver class 708 may be provided 1208 which inherits all of the data members and member functions defined by the message source class 704. It further defines data members and member functions related to protocols for receiving the message at a receiver address from the messaging service and providing the message to the client application. Alternatively, a message iterator class 710 may be provided 1208 which has data members and member functions related to iteration functionality over a collection of messages from a message source object instantiated from the message source class 704.

A file system message store name class 714 may be provided 1208 which inherits all of the data members and member functions defined by the message bin name class 716. It further defines data members and member functions related to identifying a file system message store object instantiated from the file system message store class 714.

A MessageReceiverAddress class 718 may be provided 1208 which inherits all of the data members and member functions defined by the message bin name class 716. It further defines data members and member functions related to identifying a message receiver object instantiated from the message receiver class 708. Also, a internet receiver address class 722 may be provided 1208 which inherits all of the data members and member functions defined by the MessageReceiverAddress class 718. It further defines data members and member functions related to identifying an RFC-822 compliant Internet mailbox address.

This messaging interface process may be implemented in a computer system by storing the classes in a storage device **120** and using a processor **110** in conjunction with RAM **114** and/or ROM to generate objects from the stored classes. A communications adapter **134** may communicate a message object, instantiated from the message class **724**, between the computer system and other computer systems operatively coupled together on a communication network.

In addition, a program storage device may be created which is readable by a computer system tangibly embodying a program of instructions executable by the computer system. This program of instructions would perform one or more parts of the object oriented programming based messaging interface method described about.

It is to be understood that even though numerous characteristics and advantages of various embodiments of the present invention have been set forth in the foregoing description, together with details of the structure and function of various embodiments of the invention, this disclosure is illustrative only, and changes may be made in detail, especially in matters of structure and arrangement of parts within the principles of the present invention to the full extent indicated by the broad general meaning of the terms in which the appended claims are expressed. For example, the actual names or division or functions may be changed between the OOP classes and objects, detailed above, while maintaining substantially the same functionality without departing from the scope and spirit of the present invention.

Claims

What is claimed is:

1. An object oriented programming (OOP) based messaging interface method for use between a client application and a messaging service on a computer system, comprising the steps of:
   (a) providing a message class which has data members and member functions related to a message in the messaging service;
   (b) providing a message bin class which has data members and member functions related to holding the message;
   (c) providing a message bin name class which has data members and member functions related to identifying a message bin object instantiated from the message bin class; and
   (d) generating a messaging interface object for the message as an instance of one of the provided classes, the messaging interface object furnishing the client application with protocol independent access to the messaging service.

2. The messaging interface method of claim 1 further comprising the step of providing a message address bundle class which has data members and member functions related to a message receiver address to which the message can be sent in the messaging service and related attributes of the message receiver address.

3. The messaging interface method of claim 1 further comprising the step of providing a message address iterator class which has data members and member functions related to an iterator over the addresses.

4. The messaging interface method of claim 1 further comprising the step of providing a message reference class which inherits all of the data members and member functions defined by the message class and further defines data members and member functions related to locking the message such that the message can not be modified.

5. The messaging interface method of claim 1 further comprising the step of providing a draft message class which inherits all of the data members and member functions defined by the message class and further defines data members and member functions related to data integrity and interoperability of the message.

6. The messaging interface method of claim 5 further comprising the step of providing an OOP-specific message class which inherits all of the data members and member functions defined by the draft message class and further defines data members and member functions related to guaranteeing data integrity when delivering the message to an OOP-based message system.

7. The messaging interface method of claim 5 further comprising the step of providing an interoperable message class which inherits all of the data members and member functions defined by the draft message class and further defines data members and member functions related to guaranteeing data interoperability when delivering the message to a non-OOP-based message system.

8. The messaging interface method of claim 1 further comprising the step of providing a message store class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for storing the message into a message store object instantiated from the message store class.

9. The messaging interface method of claim 8 further comprising the step of providing a message sender class which inherits all of the data members and member functions defined by the message store class and further defines data members and member functions related to protocols for submitting and sending the message from the client application to the messaging service.

10. The messaging interface method of claim 9 further comprising the steps of:

(a) providing a message source class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class; and

(b) providing a file system message store class which inherits all of the data members and member functions defined by the message sender class and the message source class and further defines data members and member functions related to protocols for storing, accessing and sending the message where the message is stored in a file system directory in a storage of the computer system.

11. The messaging interface method of claim 1 further comprising the step of providing a message source class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class.

12. The messaging interface method of claim 11 further comprising the step of providing a message receiver class which inherits all of the data members and member functions defined by the message source class and further defines data members and member functions related to protocols for receiving the message at a receiver address from the messaging service and providing the message to the client application.

13. The messaging interface method of claim 11 further comprising the step of providing a message iterator class which has data members and member functions related to iteration functionality over a collection of messages from a message source object instantiated from the message source class.

14. The messaging interface method of claim 10 further comprising the step of providing a file system message store name class which inherits all of the data members and member functions defined by the message bin name class and further defines data members and member functions related to identifying a file system message store object instantiated from the file system message store class.

15. The messaging interface method of claim 12 further comprising the step of providing a message receiver address class which inherits all of the data members and member functions defined by the message bin name class and further defines data members and member functions related to identifying a message receiver object instantiated from the message receiver class.

16. The messaging interface method of claim 15 further comprising the step of providing a internet receiver address class which inherits all of the data members and member functions defined by the message receiver address class and further defines data members and member functions related to identifying an RFC-822 compliant Internet mailbox address.

17. A program storage device readable by a computer system tangibly embodying a program of instructions executable by the computer system to perform an object oriented programming (OOP) based messaging interface method for use between a client application and a messaging service on the computer system, the method comprising the steps of:

(a) providing a message class which has data members and member functions related to a message in the messaging service;

(b) providing a message bin class which has data members and member functions related to holding the message;

(c) providing a message bin name class which has data members and member functions related to identifying a message bin object instantiated from the message bin class; and

(d) generating a messaging interface object for the message as an instance of one of the provided classes, the messaging interface object furnishing the client application with protocol independent access to the messaging service.

18. The program storage device of claim 17 wherein the method further comprises the step of providing a message address bundle class which has data members and member functions related to a message receiver address to which the message can be sent in the messaging service and related attributes of the message receiver address.

19. The program storage device of claim 17 wherein the method further comprises the step of providing a message address iterator class which has data members and member functions related to an iterator over the addresses.

20. The program storage device of claim 17 wherein the method further comprises the step of providing a message reference class which inherits all of the data members and member functions defined by the message class and further defines data members and member functions related to locking the message such that the message can not be modified.

21. The program storage device of claim 17 wherein the method further comprises the step of providing a draft message class which inherits all of the data members and member functions defined by the message class and further defines data members and member functions related to data integrity and interoperability of the message.

22. The program storage device of claim 21 wherein the method further comprises the step of providing an OOP-specific message class which inherits all of the data members and member functions defined by the draft message class and further defines data members and member functions related to guaranteeing data integrity when delivering the message to an OOP-based message system.

23. The program storage device of claim 21 wherein the method further comprises the step of providing an interoperable message class which inherits all of the data members and member functions defined by the draft message class and further defines data members and member functions related to guaranteeing data interoperability when delivering the message to a non-OOP-based message system.

24. The program storage device of claim 17 wherein the method further comprises the step of providing a message store class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for storing the message into a message store object instantiated from the message store class.

25. The program storage device of claim 24 wherein the method further comprises the step of providing a message sender class which inherits all of the data members and member functions defined by the message store class and further defines data members and member functions related to protocols for submitting and sending the message from the client application to the messaging service.

26. The program storage device of claim 25 wherein the method further comprises the steps of:

(a) providing a message source class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class; and

(b) providing a file system message store class which inherits all of the data members and member functions defined by the message sender class and the message source class and further defines data members and member functions related to protocols for storing, accessing and sending the message where the message is stored in a file system directory in a storage of the computer system.

27. The program storage device of claim 17 wherein the method further comprises the step of providing a message source class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class.

28. The program storage device of claim 27 wherein the method further comprises the step of providing a message receiver class which inherits all of the data members and member functions defined by the message source class and further defines data members and member functions related to protocols for receiving the message at a receiver address from the messaging service and providing the message to the client application.

29. The program storage device of claim 27 wherein the method further comprises the step of providing a message iterator class which has data members and member functions related to iteration functionality over a collection of messages from a message source object instantiated from the message source class.

30. The program storage device of claim 26 wherein the method further comprises the step of providing a file system message store name class which inherits all of the data members and member functions defined by the message bin name class and further defines data members and member functions related to identifying a file system message store object instantiated from the file system message store class.

31. The program storage device of claim 28 wherein the method further comprises the step of providing a message receiver address class which inherits all of the data members and member functions defined by the message bin name class and further defines data members and member functions related to identifying a message receiver object instantiated from the message receiver class.

32. The program storage device of claim 31 wherein the method further comprises the step of providing a internet receiver address class which inherits all of the data members and member functions defined by the message receiver address class and further defines data members and member functions related to identifying an RFC-822 compliant Internet mailbox address.

33. An object oriented programming (OOP) based messaging interface between a client application and a messaging service for a computer system, comprising:

(a) a storage device, in the computer system, which stores OOP-based classes, the classes, including:

(i) a message class which has data members and member functions related to a message in the messaging service;

(ii) a message bin class which has data members and member functions related to holding the message; and

(iii) a message bin name class which has data members and member functions related to identifying a message bin object instantiated from the message bin class; and

(b) a processor, operatively coupled to the storage device, which generates a messaging interface object for the message as an instance of a class stored in the storage device, the messaging interface object furnishing the client application with protocol independent access to the messaging service.

34. The messaging interface of claim 33 further comprising a communication adapter, operatively coupled to the processor, which communicates a message object, instantiated from the message class, between the computer system and other computer systems operatively coupled together on a communication network.

35. The messaging interface of claim 33 wherein the storage device further includes a message address bundle class which has data members and member functions related to a message receiver address to which the message can be sent in the messaging service and related attributes of the message receiver address.

36. The messaging interface of claim 33 wherein the storage device further includes a message address iterator class which has data members and member functions related to an iterator over the addresses.

37. The messaging interface of claim 33 wherein the storage device further includes a message reference class which inherits all of the data members and member functions defined by the message class and further defines data members and member functions related to locking the message such that the message can not be modified.

38. The messaging interface of claim 33 wherein the storage device further includes a draft message class which inherits all of the data members and member functions defined by the message class and further defines data members and member functions related to data integrity and interoperability of the message.

39. The messaging interface of claim 38 wherein the storage device further includes an OOP-specific message class which inherits all of the data members and member functions defined by the draft message class and further defines data members and member functions related to guaranteeing data integrity when delivering the message to an OOP-based message system.

40. The messaging interface of claim 38 wherein the storage device further includes an interoperable message class which inherits all of the data members and member functions defined by the draft message class and further defines data members and member functions related to guaranteeing data interoperability when delivering the message to a non-OOP-based message system.

41. The messaging interface of claim 33 wherein the storage device further includes a message store class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for storing the message into a message store object instantiated from the message store class.

42. The messaging interface of claim 41 wherein the storage device further includes a message sender class which inherits all of the data members and member functions defined by the message store class and further defines data members and member functions related to protocols for submitting and sending the message from the client application to the messaging service.

43. The messaging interface of claim 42 wherein:

(a) the storage device further includes a message source class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class; and

(b) the storage device further includes a file system message store class which inherits all of the data members and member functions defined by the message sender class and the message source class and further defines data members and member functions related to protocols for storing, accessing and sending the message where the message is stored in a file system directory in a storage of the computer system.

44. The messaging interface of claim 33 wherein the storage device further includes a message source class which inherits all of the data members and member functions defined by the message bin class and further defines data members and member functions related to protocols for retrieving the message from a message source object instantiated from the message source class.

45. The messaging interface of claim 44 wherein the storage device further includes a message receiver class which inherits all of the data members and member functions defined by the message source class and further defines data members and member functions related to protocols for receiving the message at a receiver address from the messaging service and providing the message to the client application.

46. The messaging interface of claim 44 wherein the storage device further includes a message iterator class which has data members and member functions related to iteration functionality over a collection of messages from a message source object instantiated from the message source class.

47. The messaging interface of claim 43 wherein the storage device further includes a file system message store name class which inherits all of the data members and member functions defined by the message bin name class and further defines data members and member functions related to identifying a file system message store object instantiated from the file system message store class.

48. The messaging interface of claim 45 wherein the storage device further includes a message receiver address class which inherits all of the data members and member functions defined by the message bin name class and further defines data members and member functions related to identifying a message receiver object instantiated from the message receiver class.

49. The messaging interface of claim 48 wherein the storage device further includes a internet receiver address class which inherits all of the data members and member functions defined by the message receiver address class and further defines data members and member functions related to identifying an RFC-822 compliant Internet mailbox address.

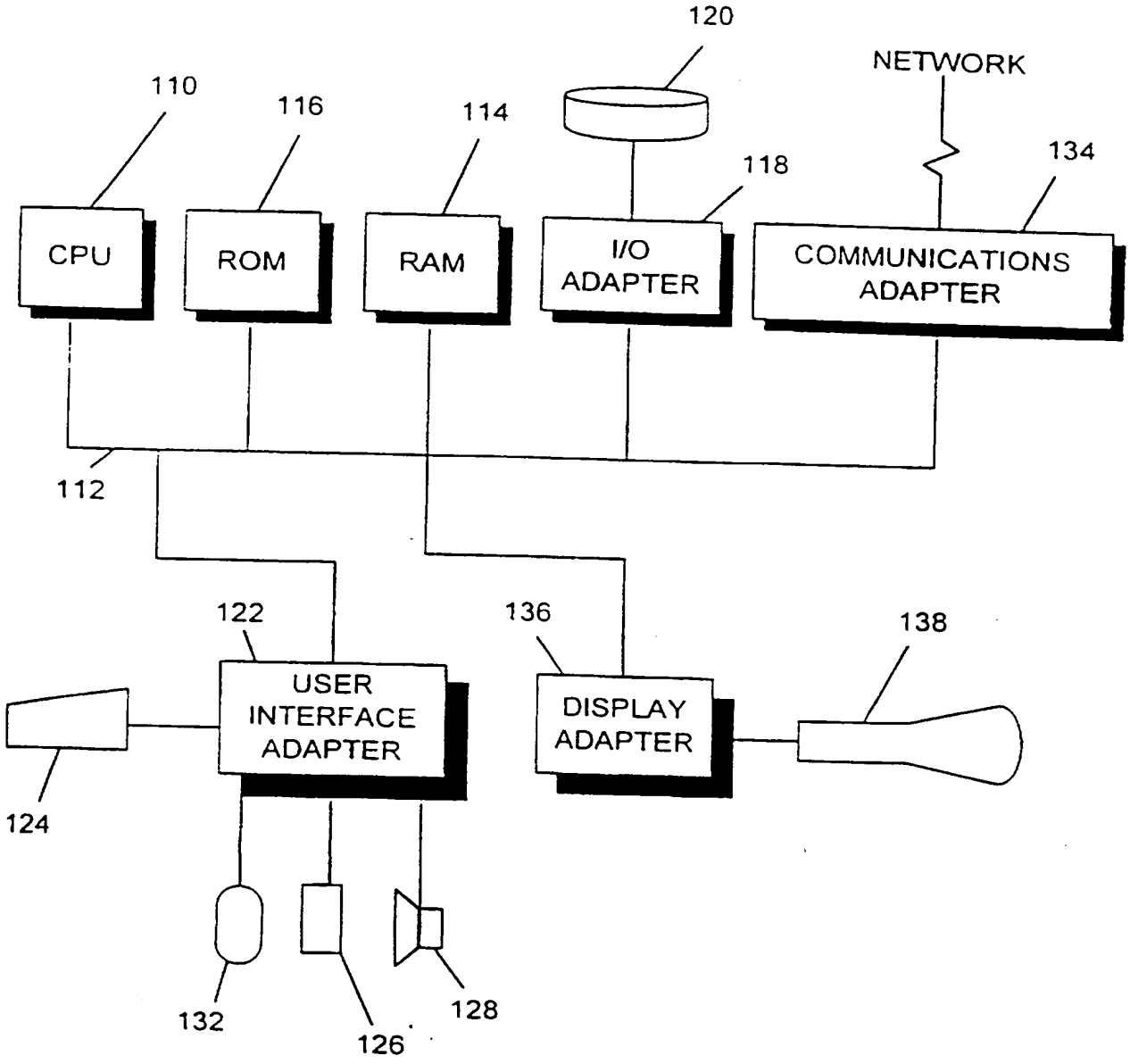


FIG. 1

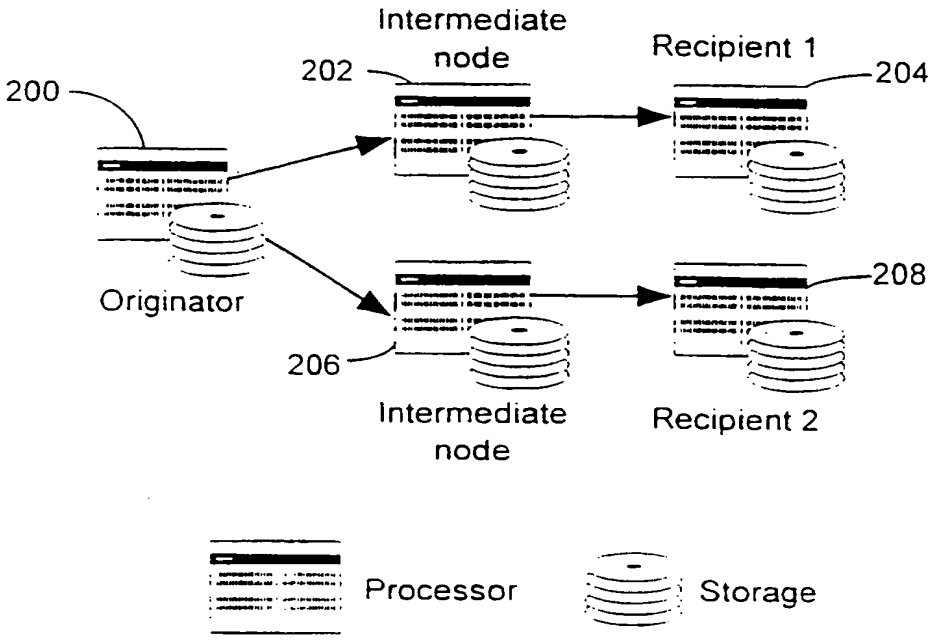


FIG. 2

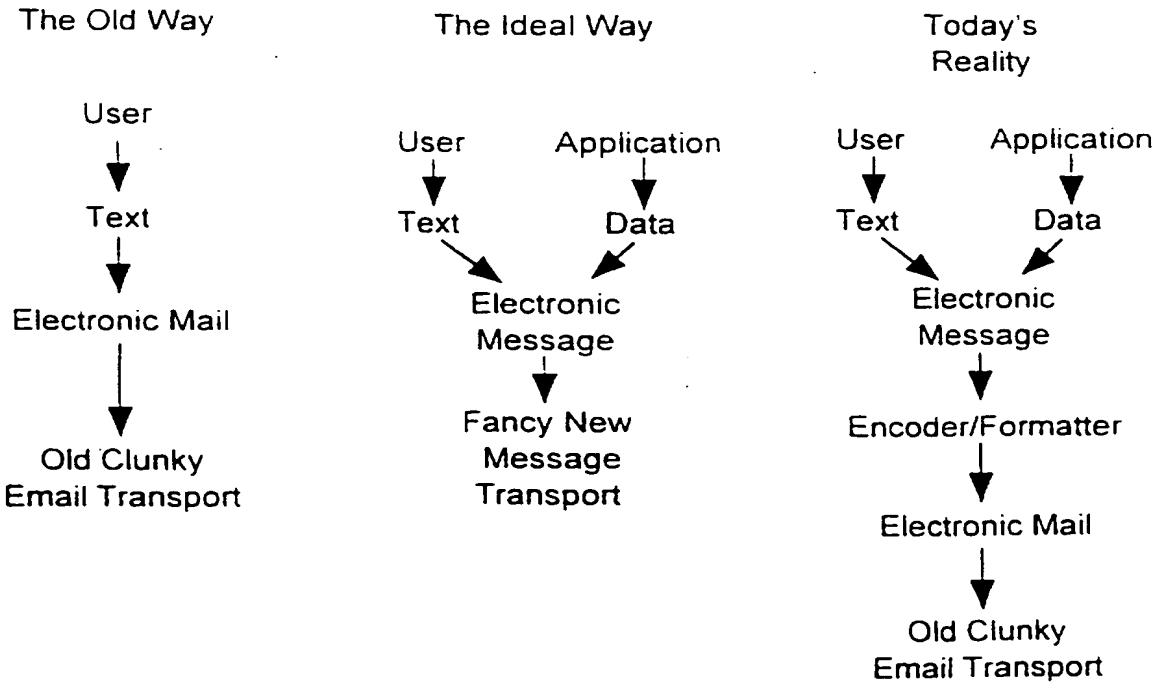


FIG. 3

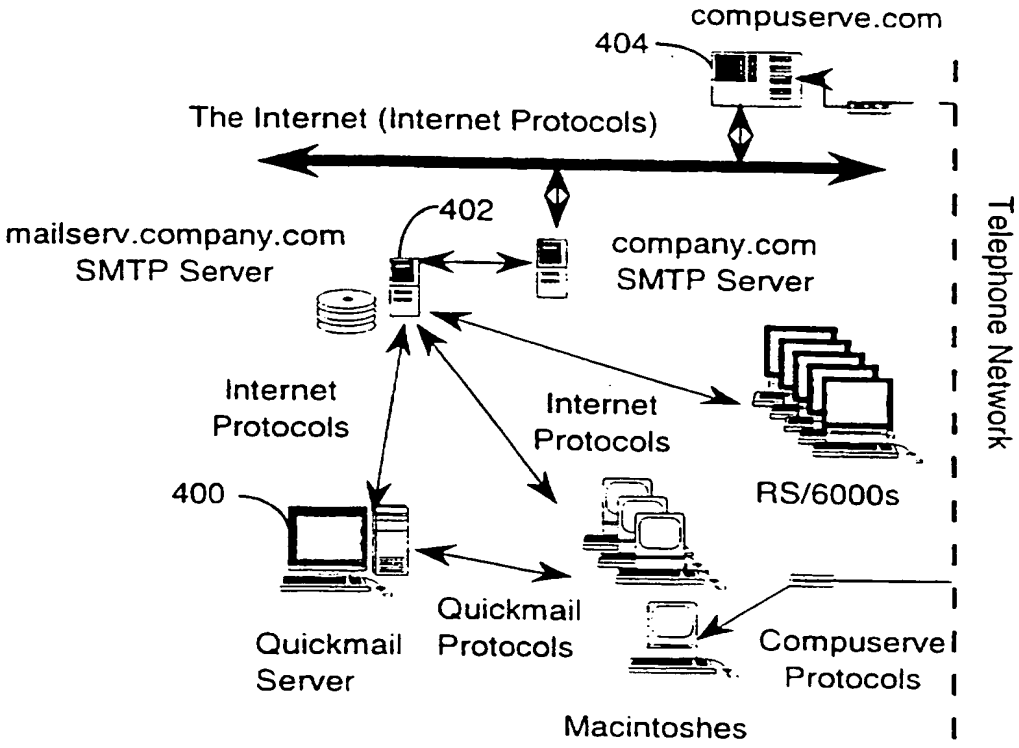


FIG. 4

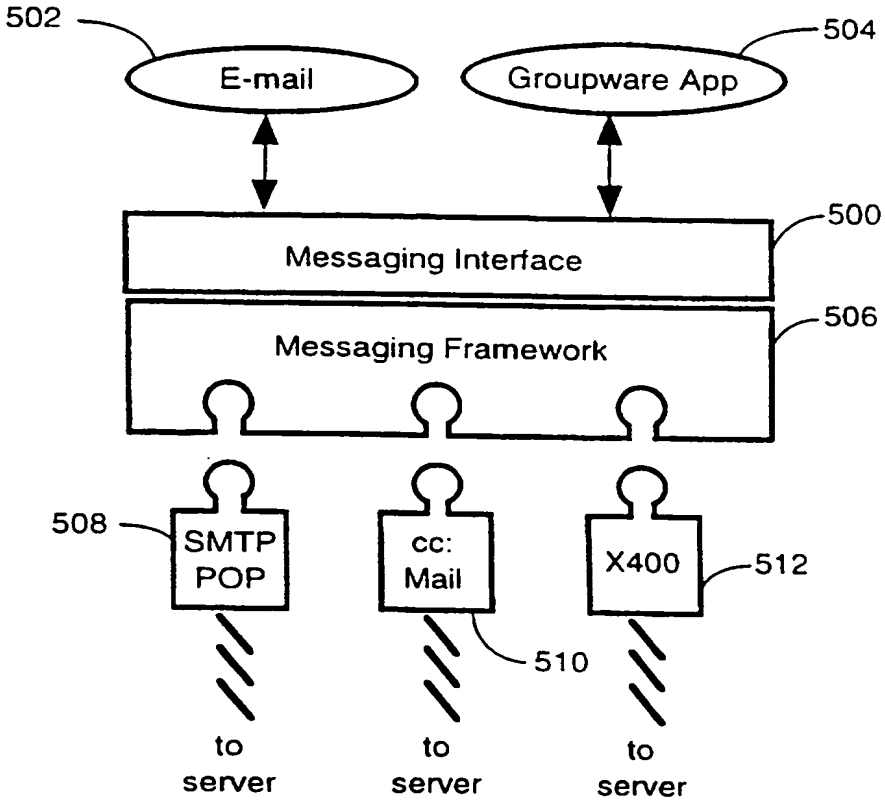


FIG. 5

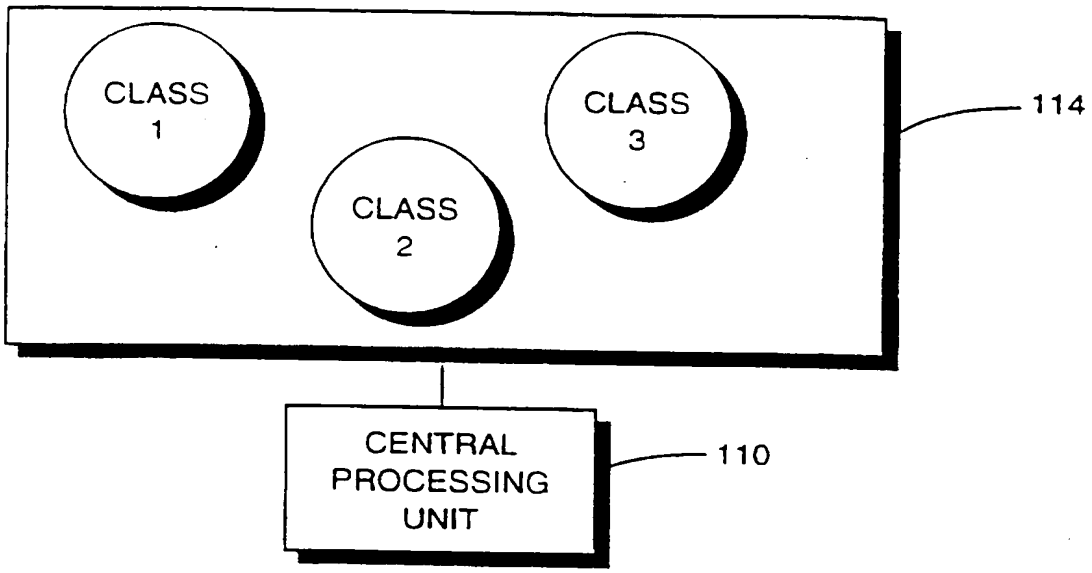


FIG. 6

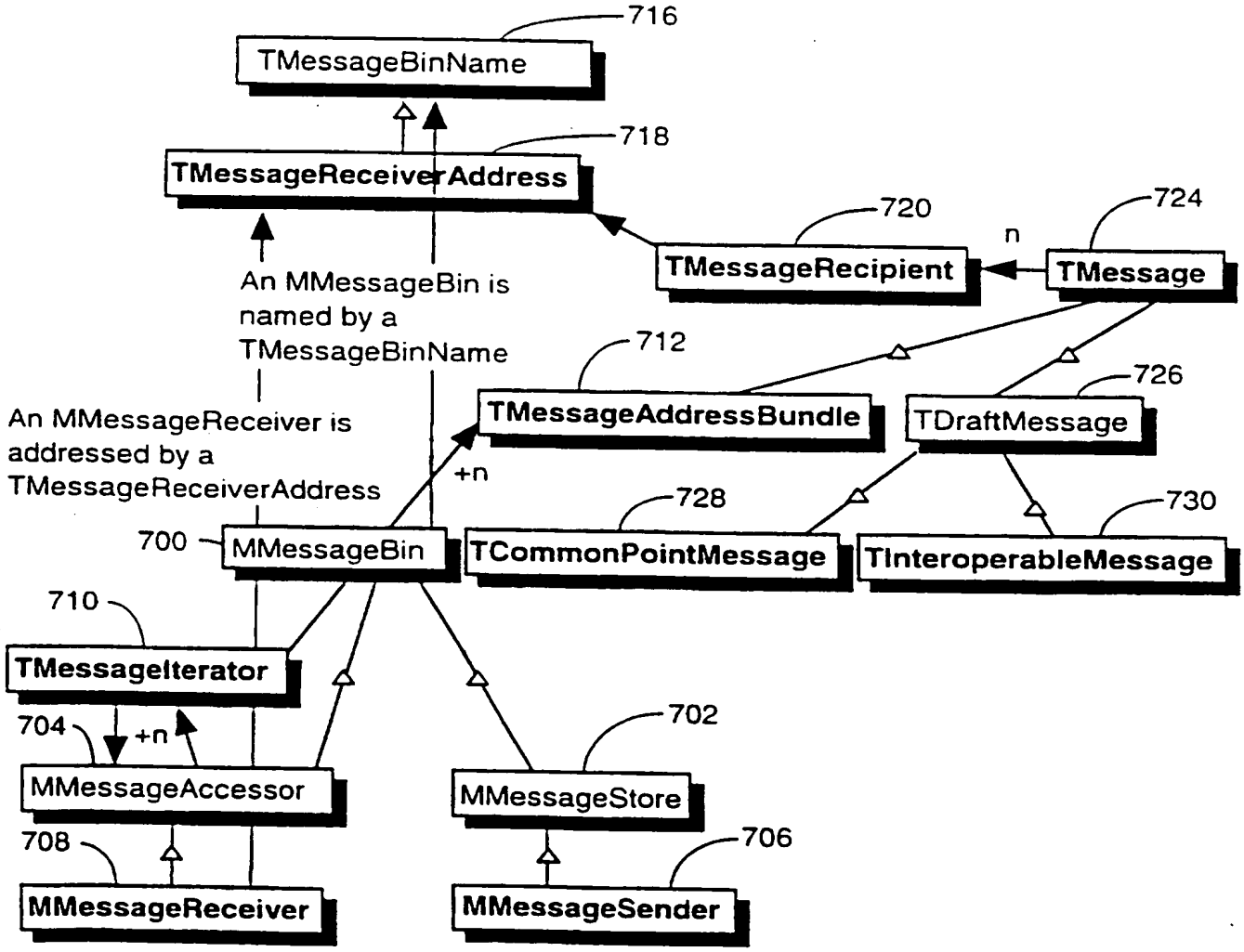


FIG. 7

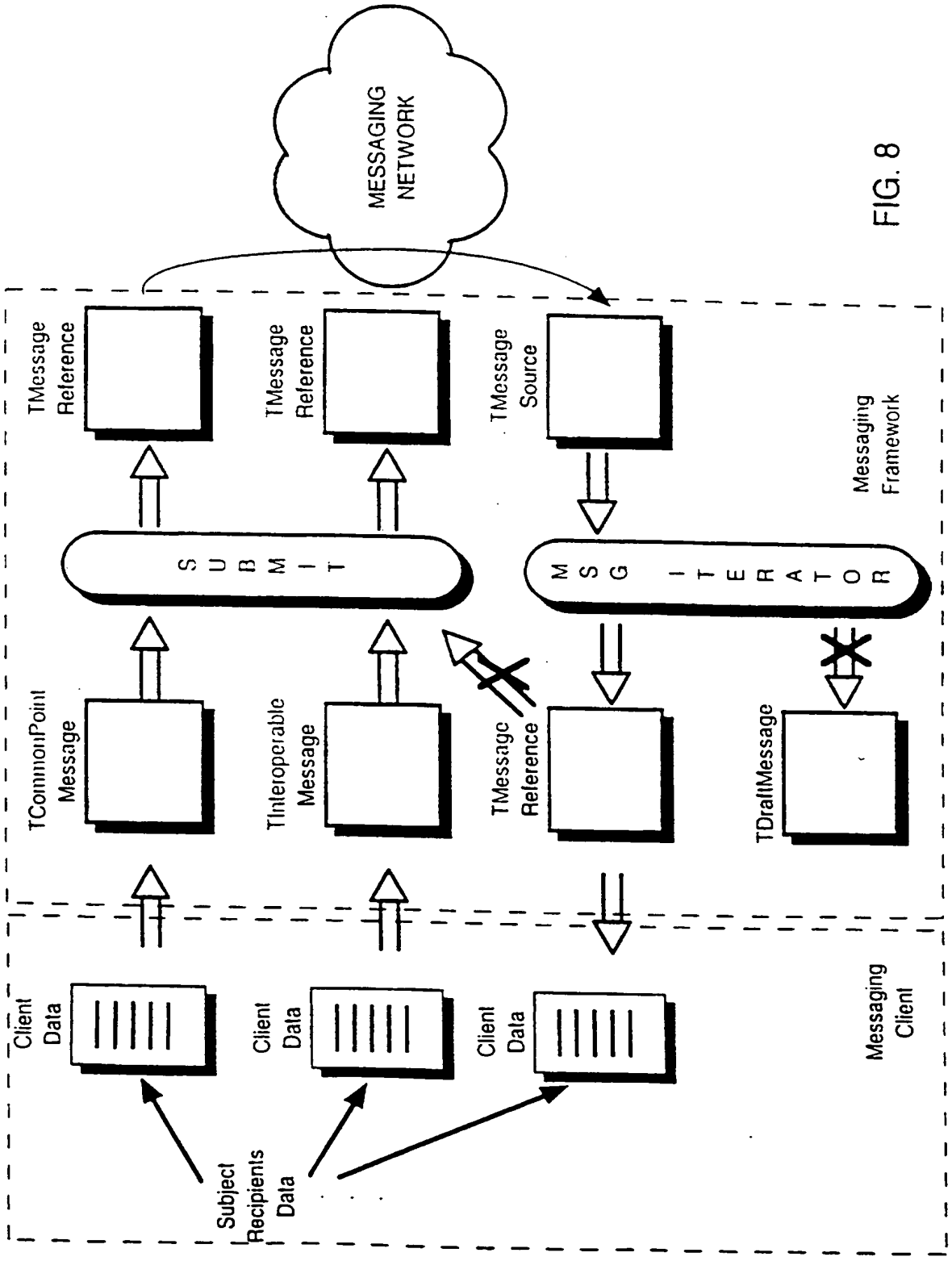


FIG. 8

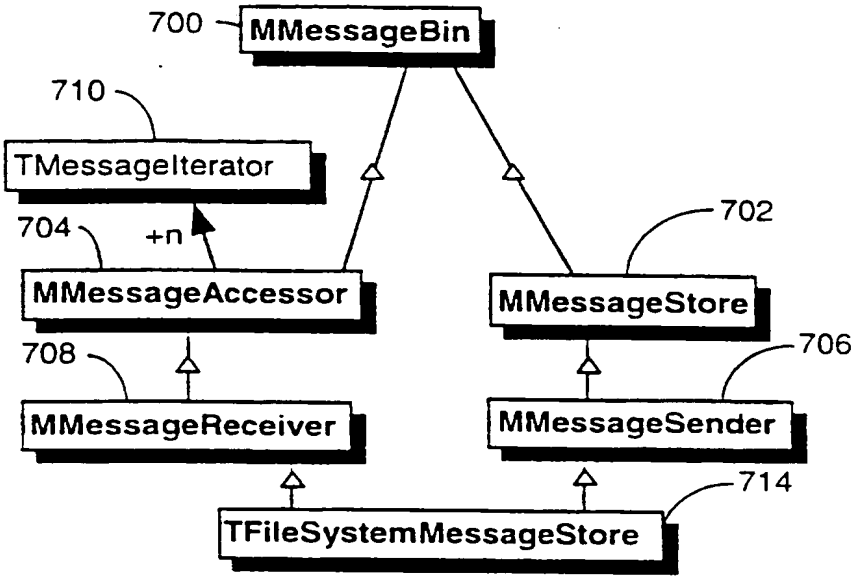


FIG. 9

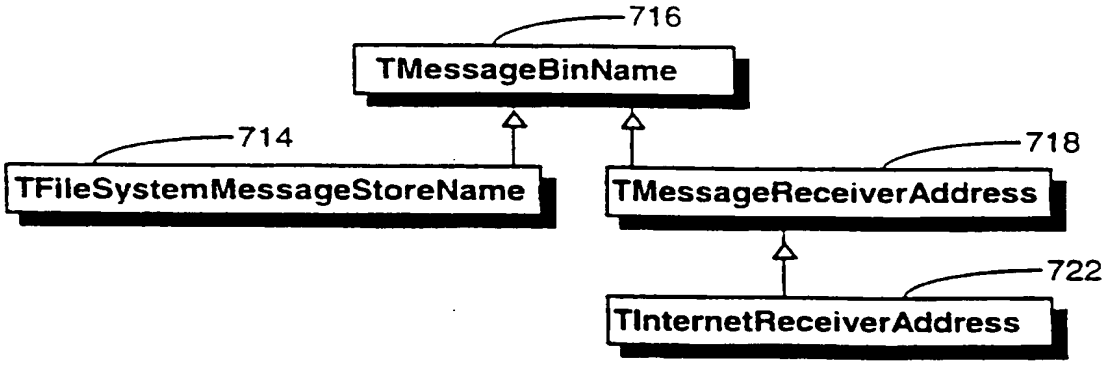


FIG. 10

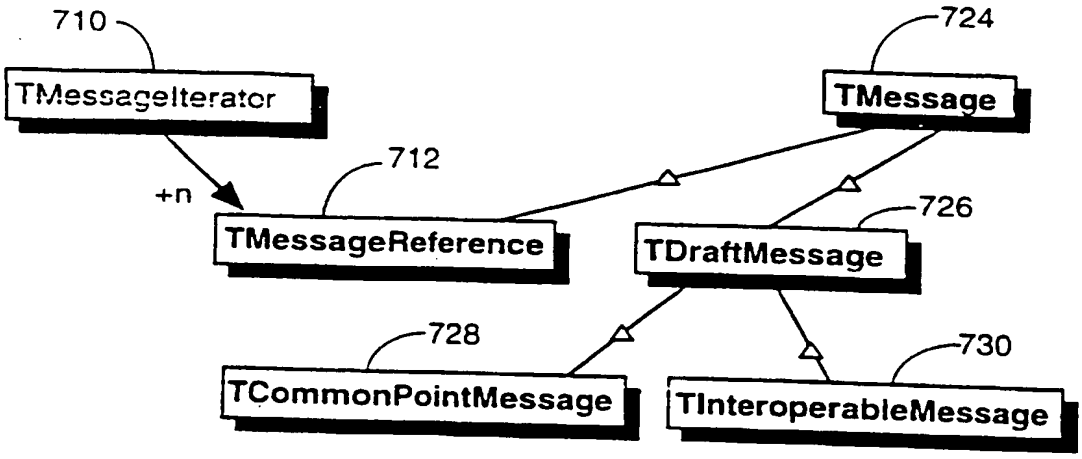


FIG. 11

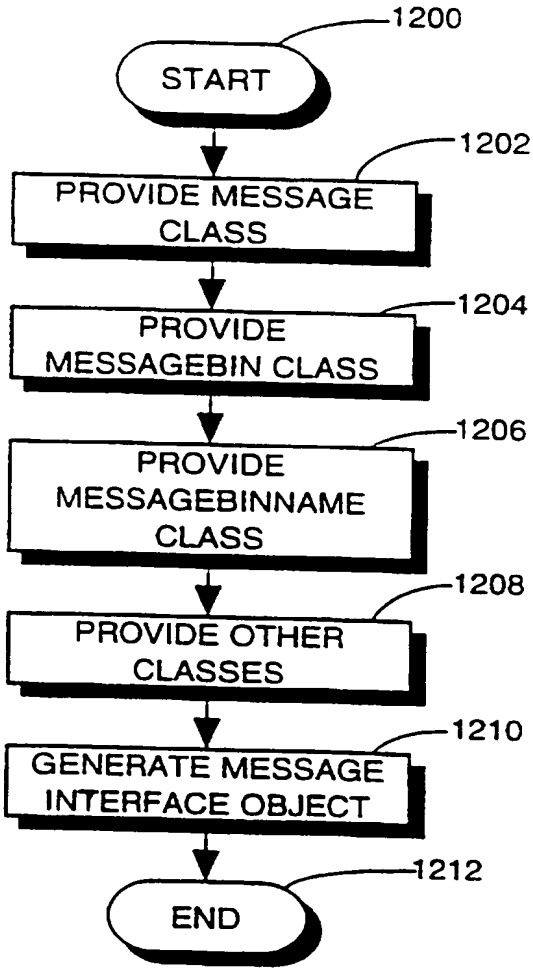


FIG. 12

INTERNATIONAL SEARCH REPORT

Inte nal Application No
PCT/US 96/20536

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	BYTE, vol. 19, no. 11, 1 November 1994, pages 163/164, 166-169, 172, 174, XP000476578 SHELDON T: "MAPI BLOOMS IN CHICAGO" see the whole document ---	1,17,33
A	SIGICE BULLETIN, vol. 20, no. 4, 1 April 1995, page 7-COMPL XP000525733 WILLIAMS P: "IBM MQSERIES COMMERCIAL MESSAGING" see page 8, line 1 - page 16, paragraph 1 see page 20, line 1 - page 21A, last line -----	1,17,33

☐ Further documents are listed in the continuation of box C.

☐ Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search
2 May 1997

Date of mailing of the international search report
22.05.97

Name and mailing address of the ISA
European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+ 31-70) 340-2040, Tx. 31 651 epo nl,
Fax: (+ 31-70) 340-3016

Authorized officer
Fonderson, A

Form PCT/ISA/210 (second sheet) (July 1992)